musicdsp.org source code archive

This PDF was autogenerated on February 14, 2004, 10:14 am, server time
All names in the first section are links, you can click'm ;-)

Analysis

Effects

Filters

- Digital RIAA equalization filter coefficients
- Formant filter
- Karlsen
- Lowpass filter for parameter edge filtering
- LP and HP filter
- Moog VCF
- Moog VCF, variation 1
- Moog VCF, variation 2
- Notch filter
- One pole LP and HP
- One pole, one zero LP/HP
- One zero, LP/HP
- Peak/Notch filter
- Phase equalization
- Pink noise filter
- Polyphase Filters
- Prewarping
- RBJ-Audio-EQ-Cookbook
- Resonant filter
- Resonant IIR lowpass (12dB/oct)
- Resonant low pass filter
- State variable
- State Variable Filter (Chamberlin version)
- State Variable Filter (Double Sampled, Stable)
- Stilson's Moog filter code
- Time domain convolution with O(n^log2(3))
- Time domain convolution with O(n^log2(3))
- Various Biquad filters
- Zoelzer biquad filters

Other

- 16-to-8-bit first-order dither
- 3rd order Spline interpollation
- 5-point spline interpollation
- Allocating aligned memory
- Antialiased Lines
- Base-2 exp
- Block/Loop Benchmarking
- Calculate notes (java)
- Center separation in a stereo mixdown
- Center separation in a stereo mixdown
- Clipping without branching
- Constant-time exponent of 2 detector
- Conversions on a PowerPC
- Copy-protection schemes
- Cubic interpollation
- Denormal DOUBLE variables, macro
- Denormal numbers
- Denormal numbers, the meta-text
- Dither code
- Dithering
- Double to Int
- Envelope Follower
- Exponential parameter mapping
- fast abs/neg/sign for 32bit floats
- Fast binary log approximations
- Fast exp2 approximation
- Fast log2
- fast power and root estimates for 32bit floats
- Float to int
- Float-to-int, coverting an array of floats
- Gaussian dithering
- Gaussian random numbers
- Hermite Interpolator (x86 ASM)
- Hermite interpollation
- Lock free fifo

- MATLAB-Tools for SNDAN
- MIDI note/frequency conversion
- Millimeter to DB (faders...)
- Noise Shaping Class
- Nonblocking multiprocessor/multithread algorithms in C++
- pow(x,4) approximation
- Reading the compressed WA! parts in gigasampler files
- Real basic DSP with Matlab (+ GUI) ...
- real value vs display value
- Really fast x86 floating point sin/cos
- resampling
- Saturation
- Sin, Cos, Tan approximation

Synthesis

- (Allmost) Ready-to-use oscillators
- Alias-free waveform generation with analog filtering
- another LFO class
- Arbitary shaped band-limited waveform generation (using oversampling and low-pass filtering)
- Audiable alias free waveform gen using width sine
- Bandlimited sawtooth synthesis
- Bandlimited waveform generation
- Bandlimited waveform generation with hard sync
- Bandlimited waveforms synopsis.
- Bandlimited waveforms...
- Cubic polynomial envelopes
- Discrete Summation Formula (DSF)
- DSF (super-set of BLIT)
- Fast LFO in Delphi...
- Fast sine and cosine calculation
- Fast sine wave calculation
- Fast square wave generator
- Gaussian White noise
- Gaussian White Noise
- Inverted parabolic envelope
- Phase modulation Vs. Frequency modulation
- Phase modulation Vs. Frequency modulation II
- Pseudo-Random generator
- Pulsewidth modulation
- quick and dirty sine generator
- SawSin
- Sine calculation
- Square Waves
- Waveform generator using MinBLEPS
- Wavetable Synthesis
- Weird synthesis

(Allmost) Ready-to-use oscillators (click this to go back to the index)

Type : waveform generation

References : Ross Bencina, Olli Niemitalo, ...

Notes :
Ross Bencina: original source code poster
Olli Niemitalo: UpdateWithCubicInterpolation

Code :
```
//this code is meant as an EXAMPLE

//uncomment if you need an FM oscillator
//define FM_OSCILLATOR

/*
members are:

float phase;
int TableSize;
float sampleRate;

float *table, dtable0, dtable1, dtable2, dtable3;

->these should be filled as folows... (remember to wrap around!!!)
table[i] = the wave-shape
dtable0[i] = table[i+1] - table[i];
dtable1[i] = (3.f*(table[i]-table[i+1])-table[i-1]+table[i+2])/2.f
dtable2[i] = 2.f*table[i+1]+table[i-1]-(5.f*table[i]+table[i+2])/2.f
dtable3[i] = (table[i+1]-table[i-1])/2.f
*/

float Oscillator::UpdateWithoutInterpolation(float frequency)
{
        int i = (int) phase;

        phase += (sampleRate/(float TableSize)/frequency;

        if(phase >= (float)TableSize)
                phase -= (float)TableSize;

#ifdef FM_OSCILLATOR
        if(phase < 0.f)
                phase += (float)TableSize;
#endif

        return table[i] ;
}

float Oscillator::UpdateWithLinearInterpolation(float frequency)
{
        int i = (int) phase;
        float alpha = phase - (float) i;

        phase += (sampleRate/(float)TableSize)/frequency;

        if(phase >= (float)TableSize)
                phase -= (float)TableSize;

#ifdef FM_OSCILLATOR
        if(phase < 0.f)
                phase += (float)TableSize;
#endif

        /*
        dtable0[i] = table[i+1] - table[i]; //remember to wrap around!!!
        */

        return table[i] + dtable0[i]*alpha;
}

float Oscillator::UpdateWithCubicInterpolation( float frequency )
{
        int i = (int) phase;
        float alpha = phase - (float) i;

        phase += (sampleRate/(float)TableSize)/frequency;

        if(phase >= (float)TableSize)
                phase -= (float)TableSize;

#ifdef FM_OSCILLATOR
        if(phase < 0.f)
                phase += (float)TableSize;
#endif

        /* //remember to wrap around!!!
        dtable1[i] = (3.f*(table[i]-table[i+1])-table[i-1]+table[i+2])/2.f
        dtable2[i] = 2.f*table[i+1]+table[i-1]-(5.f*table[i]+table[i+2])/2.f
        dtable3[i] = (table[i+1]-table[i-1])/2.f
```

```
        */
        return ((dtable1[i]*alpha + dtable2[i])*alpha + dtable3[i])*alpha+table[i];
}
```

Alias-free waveform generation with analog filtering (click this to go back to the index)

Type : waveform generation

References : Posted by Magnus Jonsson

Linked file : synthesis001.txt (this linked file is included below)

Notes :
(see linkfile)

Linked files

```
alias-free waveform generation with analog filtering
-----------------------------------------------------

Ok, here is how I did it. I'm not 100% sure that everything is correct.
I'll demonstrate it on a square wave instead, although i havent tried it.

The impulse response of an analog pole is:

r(t) = exp(p*x)   if t >= 0,
       0          if t < 0

notice that if we know r(t) we can get r(t+1) = r(t)*exp(p).

You all know what the waveform looks like. It's a constant -1
followed by constant 1 followed by .....

We need to convolve the impulse response with the waveform and sample it at discrete intervals.
What if we assume that we already know the result of the last sample?
Then we can "move" the previous result backwards by multiplying it with exp(p) since
r(t+1) = r(t)*exp(p)

Now the problem is reduced to integrate only between the last sample and the current sample, and
add that to the result.

some pseudo-code:

while forever
{
        result *= exp(pole);
        phase += freq;
        result += integrate(waveform(phase-freq*t), exp(t*pole), t=0..1);
}

integrate(square(phase-freq*t), exp(t*pole), t=0..1)

The square is constant except for when it changes sign.
Let's find out what you get if you integrate a constant multiplied with
exp(t*pole) =)

integrate(k*exp(t*pole)) = k*exp(t*pole)/pole
k = square(phase-freq*t)

and with t from 0 to 1, that becomes
k*exp(pole)/pole-k/pole = (exp(pole)-1)*k/pole

the only problem left to solve now is the jumps from +1 to -1 and vice versa.

you first calculate (exp(pole)-1)*k/pole like you'd normally do

then you detect if phase goes beyond 0.5 or 1. If so, find out exactly
where between the samples this change takes place.

subtract integrate(-2*exp(t*pole), t=0..place) from the result to
undo the error

Since I am terribly bad at explaining things, this is probably just a mess to you :)

Here's the (unoptimised) code to do this with a saw:
note that sum and pole are complex.

float getsample()
{
        sum *= exp(pole);

        phase += freq;

        sum += (exp(pole)*((phase-freq)*pole+freq)-(phase*pole+freq))/pole;

        if (phase >= 0.5)
        {
```

```
                float x = (phase-0.5)/freq;

                sum -= exp(pole*x)-1.0f;

                phase -= 1;
        }

        return sum.real();
}
```

There's big speedup potential in this i think.

Since the filtering is done "before" sampling,
aliasing is reduced, as a free bonus. with high cutoff
and high frequencies it's still audible though, but
much less than without the filter.

If aliasing is handled in some other way,
a digital filter will sound just as well, that's what i think.

-- Magnus Jonsson <zeal@mail.kuriren.nu>

another LFO class (click this to go back to the index)

References : Posted by mdsp

Linked file : LFO.zip

Notes :
This LFO uses an unsigned 32-bit phase and increment whose 8 Most Significant Bits adress a Look-up table while the 24 Least Significant Bits are used as the fractionnal part.
Note: As the phase overflow automatically, the index is always in the range 0-255.

It performs linear interpolation, but it is easy to add other types of interpolation.

Don't know how good it could be as an oscillator, but I found it good enough for a LFO.
BTW there is also different kind of waveforms.

Modifications:
We could use phase on 64-bit or change the proportion of bits used by the index and the fractionnal part.

Arbitary shaped band-limited waveform generation (using oversampling and low-pass filtering) (click this to go back to the index)

References : Posted by remage[AT]kac[DOT]poliod[DOT]hu

Code :
```
Arbitary shaped band-limited waveform generation
(using oversampling and low-pass filtering)

There are many articles about band-limited waveform synthesis techniques, that provide correct and fast
methods for generating classic analogue waveforms, such as saw, pulse, and triangle wave.  However, generating
arbitary shaped band-limited waveforms, such as the "sawsin" shape (found in this source-code archive), seems
to be quite hard using these techniques.

My analogue waveforms are generated in a _very_ high sampling rate (actually it's 1.4112 GHz for 44.1 kHz
waveforms, using 32x oversampling).  Using this sample-rate, the amplitude of the aliasing harmonics are
negligible (the base analogue waveforms has exponentially decreasing harmonics amplitudes).

Using a 511-tap windowed sync FIR filter (with Blackman-Harris window, and 12 kHz cutoff frequency) the
harmonics above 20 kHz are killed, the higher harmonics (that cause the sharp overshoot at step response) are
dampened.

The filtered signal downsampled to 44.1 kHz contains the audible (non-aliased) harmonics only.

This waveform synthesis is performed for wavetables of 4096, 2048, 1024, ... 8, 4, 2 samples.  The real-time
signal is interpolated from these waveform-tables, using Hermite-(cubic-)interpolation for the waveforms, and
linear interpolation between the two wavetables near the required note.

This procedure is quite time-consuming, but the whole waveform (or, in my implementation, the whole waveform-
set) can be precalculated (or saved at first launch of the synth) and reloaded at synth initialization.

I don't know if this is a theoretically correct solution, but the waveforms sound good (no audible aliasing).
Please let me know if I'm wrong...
```

Comments
from : Alex@smartelectronix.com
comment : Why can't you use fft/ifft
to synthesis directly wavetables of 2048,1024,..?
It'd be not so
"time consuming" comparing to FIR filtering.
Further cubic interpolation still might give you audible
distortion in some cases.
--Alex.

from : remage@kac.poliod.hu
comment : What should I use instead of cubic interpolation? (I had already some aliasing problems with cubic interpolation, but that can be solved by oversampling 4x the realtime signal generation)
Is this theory of generating waves from wavetables of 4096, 2084, ... 8, 4, 2 samples wrong?

from : Alex[AT]smartelectronix.com
comment : I think tablesize should not vary
depending on tone (4096,2048...)
and you'd better stay with the same table size for all notes (for example 4096, 4096...).

To avoid interpolation noise
(it's NOT caused by aliasing)
try to increase wavetable size
and be sure that waveform spectrum has
steep roll off
(don't forget Gibbs phenomena as well).

Audiable alias free waveform gen using width sine (click this to go back to the index)

Type : Very simple

References : Posted by joakim.dahlstrom@ongame.com

Notes :
Warning, my english abilities is terribly limited.

How ever, the other day when finally understanding what bandlimited wave creation is (i am a noobie, been doing DSP stuf on and off for a half/year) it hit me i can implement one little part in my synths. It's all about the freq (that i knew), very simple you can reduce alias (the alias that you can hear that is) extremely by keeping track of your frequence, the way i solved it is using a factor, afact = 1 - sin(f*2PI). This means you can do audiable alias free synthesis without very complex algorithms or very huge tables, even though the sound becomes kind of low-filtered.
Propably something like this is mentioned b4, but incase it hasn't this is worth looking up

The psuedo code describes it more.

// Druttis

Code :
```
f := freq factor, 0 - 0.5 (0 to half samplingrate)

afact(f) = 1 - sin(f*2PI)

t := time (0 to ...)
ph := phase shift (0 to 1)
fm := freq mod (0 to 1)

sine(t,f,ph,fm) = sin((t*f+ph)*2PI + 0.5PI*fm*afact(f))

fb := feedback (0 to 1) (1 max saw)

saw(t,f,ph,fm,fb) = sine(t,f,ph,fb*sine(t-1,f,ph,fm))

pm := pulse mod (0 to 1) (1 max pulse)
pw := pulse width (0 to 1) (1 square)

pulse(t,f,ph,fm,fb,pm,pw) = saw(t,f,ph,fm,fb) - (t,f,ph+0.5*pw,fm,fb) * pm

I am not completely sure about fm for saw & pulse since i cant test that atm. but it should work :) otherwise
just make sure fm are 0 for saw & pulse.

As you can see the saw & pulse wave are very variable.

// Druttis
```

Comments
from : druttis@darkface.pp.se
comment : Um, reading it I can see a big flaw...

afact(f) = 1 - sin(f*2PI) is not correct!

should be

afact(f) = 1 - sqrt(f * 2 / sr)

where sr := samplingrate
f should be exceed half sr

from : laurent@ohmforce.com
comment :

from : laurent@ohmforce.com
comment : f has already be divided by sr, right ? So it should become :

afact (f) = 1 - sqrt (f * 2)

And i see a typo (saw forgotten in the second expression) :

pulse(t,f,ph,fm,fb,pm,pw) = saw(t,f,ph,fm,fb) - saw(t,f,ph+0.5*pw,fm,fb) * pm

However I haven't checked the formula.

from : 909@gmx.de
comment : Hi Lauent,
I'm new to that DSP stuff and can't get the key to
what'S the meaning of afact? - Can you explain please!? - Thanks in advice!

Bandlimited sawtooth synthesis (click this to go back to the index)

Type : DSF BLIT

References : Posted by emanuel.landeholm [AT] telia.com

Linked file : synthesis002.txt (this linked file is included below)

Notes :
This is working code for synthesizing a bandlimited sawtooth waveform. The algorithm is DSF BLIT + leaky integrator. Includes driver code.

There are two parameters you may tweak:

1) Desired attenuation at nyquist. A low value yields a duller sawtooth but gets rid of those annoying CLICKS when sweeping the frequency up real high. Must be strictly less than 1.0!

2) Integrator leakiness/cut off. Affects the shape of the waveform to some extent, esp. at the low end. Ideally you would want to set this low, but too low a setting will give you problems with DC.

Have fun!
/Emanuel Landeholm

(see linked file)

Comments
from : rainbow_stash@hotmail.com
comment :
there is no need to use a butterworth design for a simple leaky integrator, in this case actually the variable curcps can be used directly in a simple: leak += curcps * (blit - leak);

this produces a nearly perfect saw shape in almost all cases

Linked files

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Bandlimited synthesis of sawtooth by
 * leaky integration of a DSF BLIT
 *
 * Emanuel Landeholm, March 2002
 * emanuel.landeholm@telia.com
 *
 * Provided \"as is\".
 * Free as in Ef Are Ee Ee.
 */

double pi    = 3.1415926535897932384626433832795029L;
double twopi = 6.2831853071795864769252867665590058L;

/* Leaky integrator/first order lowpass which
 * shapes the impulse train into a nice
 * -6dB/octave spectrum
 *
 * The cutoff frequency needs to be pretty lowish
 * or the sawtooth will suffer phase distortion
 * at the low end.
 */

typedef struct
{
  double x1, y1;
  double a, b;
} lowpass_t;

/* initializes a lowpass, sets cutoff/leakiness */

void init_lowpass(lowpass_t *lp, double cutoff)
{
  double Omega;

  lp->x1 = lp->y1 = 0.0;

  Omega = atan(pi * cutoff);
  lp->a = -(1.0 - Omega) / (1.0 + Omega);
  lp->b = (1.0 - lp->b) / 2.0;
}

double update_lowpass(lowpass_t *lp, double x)
{
```

```
  double y;

  y = lp->b * (x + lp->x1) - lp->a * lp->y1;

  lp->x1 = x;
  lp->y1 = y;

  return y;
}

/* dsf blit datatype
 *
 */

typedef struct
{
  double phase;    /* phase accumulator */
  double aNQ;      /* attenuation at nyquist */
  double curcps;   /* current frequency, updated once per cycle */
  double curper;   /* current period, updated once per cycle */
  lowpass_t leaky; /* leaky integrator */
  double N;        /* # partials */
  double a;        /* dsf parameter which controls roll-off */
  double aN;       /* former to the N */
} blit_t;

/* initializes a blit structure
 *
 * The aNQ parameter is the desired attenuation
 * at nyquist. A low value yields a duller
 * sawtooth but gets rid of those annoying CLICKS
 * when sweeping the frequency up real high. |aNQ|
 * must be strictly less than 1.0! Find a setting
 * which works for you.
 *
 * The cutoff parameter controls the leakiness of
 * the integrator.
 */

void init_blit(blit_t *b, double aNQ, double cutoff)
{
  b->phase  = 0.0;
  b->aNQ = aNQ;
  b->curcps = 0.0;
  b->curper = 0.0;
  init_lowpass(&b->leaky, cutoff);
}

/* Returns a sawtooth computed from a leaky integration
 * of a DSF bandlimited impulse train.
 *
 * cps (cycles per sample) is the fundamental
 * frequency: 0 -> 0.5 == 0 -> nyquist
 */

double update_blit(blit_t *b, double cps)
{
  double P2, beta, Nbeta, cosbeta, n, d, blit, saw;

  if(b->phase >= 1.0 || b->curcps == 0.0)
    {
      /* New cycle, update frequency and everything
       * that depends on it
       */

      if(b->phase >= 1.0)
b->phase -= 1.0;

      b->curcps = cps;        /* this cycle\'s frequency */
      b->curper = 1.0 / cps;  /* this cycle\'s period */

      P2 = b->curper / 2.0;
      b->N = 1.0 + floor(P2); /* # of partials incl. dc */

      /* find the roll-off parameter which gives
       * the desired attenuation at nyquist
       */

      b->a   = pow(b->aNQ, 1.0 / P2);
      b->aN  = pow(b->a,   b->N);
    }

  beta = twopi * b->phase;
```

```
  Nbeta = b->N * beta;
  cosbeta = cos(beta);

  /* The dsf blit is scaled by 1 / period to give approximately the same
   * peak-to-peak over a wide range of frequencies.
   */

  n = 1.0 -
    b->aN * cos(Nbeta) -
    b->a * (cosbeta - b->aN * cos(Nbeta - beta));
  d = b->curper * (1.0 + b->a * (-2.0 * cosbeta + b->a));

  b->phase += b->curcps; /* update phase */

  blit = n / d - b->curcps; /* This division can only fail if |a| == 1.0
     * Subtracting the fundamental frq rids of DC
     */

  saw = update_lowpass(&b->leaky, blit); /* shape blit spectrum into a saw */

  return saw;
}

/* driver code - writes headerless 44.1 16 bit PCM to stdout */

static int clipped = 0;

static void ADC_out(double x)
{
  short s;

  if(x > 1.0)
    {
      ++clipped;
      x = 1.0;
    }
  else if(x < -1.0)
    {
      ++clipped;
      x = -1.0;
    }

  s = 32767.0 * x;

  fwrite(&s, sizeof(s), 1, stdout);
}

int main(int argc, char **argv)
{
  int i, L;
  double x, cps, cm;
  blit_t b;

  L = 1000000;

  init_blit(&b, 0.5, 0.0001);

  /* sweep from 40 to 20000 Hz */

  cps = 40.0 / 44100.0;
  cm = pow(500.0, 1.0 / (double)L);

  for(i = 0; i < L; ++i)
    {
      x = 2.0 * update_blit(&b, cps);
      ADC_out(x);

      cps *= cm;
    }

  fprintf(stderr, \"%d values were clipped\\n\", clipped);

  return 0;
}
```

Bandlimited waveform generation (click this to go back to the index)

Type : waveform generation

References : Posted by Joe Wright

Linked file : bandlimited.cpp (this linked file is included below)

Linked file : bandlimited.pdf

Notes :
(see linkfile)

Linked files

```
// An example of generating the sawtooth and parabola wavetables
// for storage to disk.
//
// SPEED=sampling rate, e.g. 44100.0f
// TUNING=pitch of concert A, e.g. 440.0f

///////////////////////////
// Wavetable reverse lookup
// Given a playback rate of the wavetable, what is wavetables index?
//
// rate = f.wavesize/fs e.g. 4096f/44100
// max partials = nyquist/f = wavesize/2rate  e.g. 2048/rate
//
// using max partials we could then do a lookup to find the wavetables index
// in a pre-calculated table
//
// however, we could skip max partials, and lookup a table based on a
// function of f (or rate)
//
// the first few midi notes (0 - 9) differ by < 1 so there are duplicates
// values of (int) f.
// therefore, to get an index to our table (that indexes the wavetables)
// we need 2f
//
// to get 2f from rate we multiply by the constant
// 2f = 2.fs/wavesize  e.g. 88200/4096
//
// our lookup table will have a length>25087 to cover the midi range
// we'll make it 32768 in length for easy processing


 int a,b,n;
 float* data;
 float* sinetable=new float[4096];
 float* datap;
 for(b=0;b<4096;b++)
  sinetable[b]=sin(TWOPI*(float)b/4096.0f);
 int partials;
 int partial;
 int partialindex,reverseindex,lastnumpartials;
 float max,m;
 int* reverse;

 // sawtooth

  data=new float[128*4096];
  reverse=new int[32768];

  reverseindex=0;
  partialindex=0;
  lastnumpartials=-1;

  for(n=0;n<128;n++)
  {
   partials=(int)((SPEED*0.5f)/float(TUNING*(float)pow(2,(float) (n-69)/12.0f))); //(int) NYQUIST/f
   if(partials!=lastnumpartials)
   {
    datap=&data[partialindex*4096];
    for(b=0;b<4096;b++)
     datap[b]=0.0f;  //blank wavetable
    for(a=0;a<partials;a++)
    {
     partial=a+1;
     m=cos((float)a*HALFPI/(float)partials);  //gibbs
     m*=m;  //gibbs
     m/=(float)partial;
     for(b=0;b<4096;b++)
      datap[b]+=m*sinetable[(b*partial)%4096];
```

```
    }
    lastnumpartials=partials;
    a=int(2.0f*TUNING*(float)pow(2,(float) (n-69)/12.0f)); //2f
    for(b=reverseindex;b<=a;b++)
     reverse[b]=partialindex;
    reverseindex=a+1;
    partialindex++;
   }
  }

  for(b=reverseindex;b<32768;b++)
   reverse[b]=partialindex-1;

  ar << (int) partialindex; //number of waveforms
  ar << (int) 4096; //waveform size (in samples)

  max=0.0;
  for(b=0;b<4096;b++)
  {
   if(fabs(*(data+b))>max)  //normalise to richest waveform (0)
    max=(float)fabs(*(data+b));
  }
  for(b=0;b<4096*partialindex;b++)
  {
   *(data+b)/=max;
  }

  //ar.Write(data,4096*partialindex*sizeof(float));
  //ar.Write(reverse,32768*sizeof(int));

  delete [] data;
  delete [] reverse;
 }
 // end sawtooth

 // parabola

  data=new float[128*4096];
  reverse=new int[32768];

  reverseindex=0;
  partialindex=0;
  lastnumpartials=-1;

  float sign;

  for(n=0;n<128;n++)
  {
   partials=(int)((SPEED*0.5f)/float(TUNING*(float)pow(2,(float) (n-69)/12.0f)));
   if(partials!=lastnumpartials)
   {
    datap=&data[partialindex*4096];
    for(b=0;b<4096;b++)
     datap[b]=PI*PI/3.0f;
    sign=-1.0f;
    for(a=0;a<partials;a++)
    {
     partial=a+1;
     m=cos((float)a*HALFPI/(float)partials); //gibbs
     m*=m;  //gibbs
     m/=(float)(partial*partial);
     m*=4.0f*sign;
     for(b=0;b<4096;b++)
      datap[b]+=m*sinetable[((b*partial)+1024)%4096]; //note, parabola uses cos
     sign=-sign;
    }
    lastnumpartials=partials;
    a=int(2.0f*TUNING*(float)pow(2,(float) (n-69)/12.0f)); //2f
    for(b=reverseindex;b<=a;b++)
     reverse[b]=partialindex;
    reverseindex=a+1;
    partialindex++;
   }
  }

  for(b=reverseindex;b<32768;b++)
   reverse[b]=partialindex-1;

  ar << (int) partialindex; //number of waveforms
  ar << (int) 4096; //waveform size (in samples)

  max=0.0;
  for(b=0;b<4096;b++)
```

```
  {
   if(fabs(*(data+b))>max)  //normalise to richest waveform (0)
    max=(float)fabs(*(data+b));
  }
  max*=0.5;
  for(b=0;b<4096*partialindex;b++)
  {
   *(data+b)/=max;
   *(data+b)-=1.0f;
  }

  //ar.Write(data,4096*partialindex*sizeof(float));
  //ar.Write(reverse,32768*sizeof(int));

  delete [] data;
  delete [] reverse;
 }
 // end parabola


/////////////////////////////////////////////////////////////////////////
// An example of playback of a sawtooth wave
// This is not optimised for easy reading
// When optimising you'll need to get this in assembly (especially those
// float to int conversions)
/////////////////////////////////////////////////////////

#define WAVETABLE_SIZE  (1 << 12)
#define WAVETABLE_SIZEF  WAVETABLE_SIZE.0f
#define WAVETABLE_MASK  (WAVETABLE_SIZE - 1)

float index;
float rate;
int wavetableindex;
float ratetofloatfactor;
float* wavetable;

void setupnote(int midinote /*0 - 127*/)
{
 float f=TUNING*(float)pow(2,(float) (midinote-69)/12.0f));
 rate=f*WAVETABLE_SIZEF/SPEED;
 ratetofloatfactor=2.0f*SPEED/WAVETABLE_SIZEF;
 index=0.0f;
 wavetableindex=reverse[(int)(2.0f*f)];
 wavetable=&sawtoothdata[wavetableindex*WAVETABLE_SIZE];
}

void generatesample(float* buffer,int length)
{
 int currentsample,
 int nextsample;
 float m;
 float temprate;
 while(length--)
 {
  currentsample=(int) index;
  nextsample=(currentsample+1) & WAVETABLE_MASK;
  m=index-(float) currentsample; //fractional part
  *buffer++=(1.0f-m)*wavetable[currentsample]+m*wavetable[nextsample]; //linear interpolation
  rate*=slide; //slide coeffecient if required
  temprate=rate*fm; //frequency modulation if required
  index+=temprate;
  if(index>WAVETABLE_SIZEF)
  {
   //new cycle, respecify wavetable for sliding
   wavetableindex=reverse[(int)(ratetofloatfactor*temprate)];
   wavetable=&sawtoothdata[wavetableindex*WAVETABLE_SIZE];
   index-=WAVETABLE_SIZEF;
  }
 }
}
```

Bandlimited waveform generation with hard sync (click this to go back to the index)

References : Posted by Emanuel Landeholm

Linked file : http://www.algonet.se/~e-san/hardsync.tar.gz

Bandlimited waveforms synopsis. (click this to go back to the index)

References : Joe Wright

Linked file : waveforms.txt (this linked file is included below)

Notes :
(see linkfile)

Linked files

```
From: "Joe Wright" <joe@nyrsound.com>
To: <music-dsp@shoko.calarts.edu>
Subject: Re: waveform discussions
Date: Tue, 19 Oct 1999 14:45:56 +0100

After help from this group and reading various literature I have finished my
waveform engine.  As requested, I am now going to share some of the things
are have learnt from a practical viewpoint.

Problem:

The waveforms of interest are sawtooth, square and triangle.

The waveforms must be bandlimited (i.e. fitting under Nyquist).  This
precludes simple generation of the waveforms.  For example, the
analogue/continuous formular (which has infinite harmonics):

s(t) = (t*f0) mod 1  (t=time, f0=frequency of note)

procudues aliasing that cannot be filtered out when converted to the
digital/discrete form:

s(n) = (f0*n*Ts) mod 1 (n=sample, Ts equals 1/sampling rate)


The other condition of this problem is that the waveforms are generatable in
real-time.  Additionally, bonuses are given for solutions which allow
pulse-width modulation and triangle asymmetry.


The generation of these waves is non-triaval and below is discussed three
techniques to solve the problem - wavetables, approximation through
processing of |sinewave| and BLIT integration.  BLIT integration is
discussed in depth.


Wavetables:

You can generate a wavetable for the waveform by summing sine waves up to
the Nyquist frequency.  For example, the sawtooth waveform can be generated
by:

s(n) = Sum[k=1,n] (1/k *sin(2PI*f0*k*Ts)) where f0*n < Ts/2

The wavetable can then be played back, pitched up or down subject that
pitched f*n < Ts/2.  Anything lower will not alias but it may lack some
higher harmonics if pitched too low.

To cover these situations, use multiple wavetables describing different
frequency ranges within which it is fine to pitch up or down.

You may need to compromise between number of wavetables and accuracy because
of memory considerations (especially if over-sampling).  This means some
wavetables will have to cover larger ranges than they should.  As long as
the range is too far in the lower direction rather than higher, you will not
alias (you will just miss some higher harmonics).

With wavetables you can add a sawtooth to an inverted sawtooth offset in
time, to produce a pulse/square wave.  Vary the offset to vary the pulse
width.  Asymmetry of triangle waves is not possible (as far as I know)
though.


Approximation through processing of |sinewave|

This method is discussed in detail in 'Modeling Analog Synthesis with
DSPs' - Computer Music Journal, 21:4 pp.23 - 41, Winter 1997, Lane, Hoory,
Marinez and Wang.

The basic idea starts with the generation of a sawtooth by feeding
abs(sin(n)) into a lowpass followed by a highpass filter.  This approximates
(quite well supposedly) a bandlimited sawtooth.  Unfortunately, details of
```

what the cutoff for the lowpass should be were not precise in the paper (although the highpass cutoff was given).

Square wave and triangle wave approximation was implemented by subtracting abs(sin(n)) and abs(sin(n/2)). With different gains, pulse width (and I presume asymmetry) were possible.

For more information, consult the paper.

BLIT intergration

This topic refers to the paper 'Alias-Free Digital Synthesis of Classic Analog Waveforms' by Tim Stilson and Julius Smith of CCRMA. The paper can be found at http://www-ccrma.stanford.edu/~stilti/papers

BLIT stands for bandlimited impluse train. I'm not going to go into the theory, you'll have to read the paper for that. However, simply put, a pulse train of the form 10000100001000 etc... is not bandlimited.

The formular for a BLIT is as follows:

BLIT(x) = (m/p) * (sin(PI*x*m/p)/(m*sin(PI*x/p))

x=sample number ranging from 1 to period
p=period in samples (fs/f0). Although this should theorectically not be an interger, I found pratically speaking it needs to be.
m=2*((int)p/2)+1 (i.e. when p=odd, m=p otherwise m=p+1)

[note] in the paper they describe m as the largest odd interger not exceeding the period when in fact their formular for m (which is the one that works in practive) makes it (int)p or (int)p +1

As an extension, we also have a bipolar version:

BP-BLIT k (x)=BLIT(x) - BLIT(x+k)   (where k is in the range [0,period])

Now for the clever bit. Lets start with the square/rectangle wave. Through intergration we get:

Rect(n) = Sum(i=0,n) (BP-BLIT k0 (i) - C4)

C4 is a DC offset which for the BP-BLIT is zero. This gives a nice iteration for rect(n):

Rect(n) = Rect(n-1) + BP-BLIT k0 (n) where k0 is the pulse width between [0,period] or in practive [1,period-1]

A triangle wave is given by:

Tri(n) = Sum(i=0,n) (Rect(k) - C6)

Tri(n) = Tri(n-1) + Rect(n) - C6

C6 = k0/period

The triangle must also be scaled:

Tri(b) = Tri(n-1) + g(f,d)*(Rect(n) - C6)

where g(f,d) = 0.99 / (period* d * (d-1))   d=k0/period

Theorietcally it could be 1.00 / ... but I found numerical error sometimes pushed it over the edge. The paper actually states 2.00/... but for some reason I find this to be incorrect.

Lets look at some rough and ready code. I find the best thing to do is to generate one period at a time and then reset everything. The numerical errors over one period are negligable (based on 32bit float) but if you keep on going without resetting, the errors start to creep in.

```
    float period = samplingrate/frequency;
    float m=2*(int)(period/2)+1.0f;
    float k=(int)(k0*period);
    float g=0.99f/(periodg*(k/period)*(1-k/period));
    float t;
    float bpblit;
    float square=0.0f;
    float triangle=0.0f;
    for(t=1;t<=period;t++)
    {
```

```
        bpblit=sin(PI*(t+1)*m/period)/(m*sin(PI*(t+1)/period));
        bpblit-=sin(PI*(t+k)*m/period)/(m*sin(PI*(t+k)/period));
        square+=bpblit;
        triangle+=g*(square+k/period);
    }
    square=0;
    triangle=0;
```

Highly un-optimised code but you get the point. At each sample(t) the output values are square and triangle respectively.

Sawtooth:

This is given by:

```
Saw(n) = Sum(k=0,n) (BLIT(k) - C2)   [as opposed to BP-BLIT]
Saw(n) = Saw(n-1) + BLIT(n) - C2
```

Now, C2 is a bit tricky. Its the average of BLIT for that period (i.e. 1/n * Sum (k=0,n) (BLIT(k)). [Note] Blit(0) = 0.

I found the best way to deal with this is to have a lookup table which you have generated and saved to disk as a file which contains a value of C2 for every period you are interested in. This is because I know of no easy way to generate C2 in real-time.

Last thing. My implementation of BLIT gives negative values. Therefore my sawtooth is +C2 rather than -C2.

I hope this helps, any questions don't hestitate to contact me.

Joe Wright - Nyr Sound Ltd
http://www.nyrsound.com
info@nyrsound.com

Bandlimited waveforms... (click this to go back to the index)

References : Posted by Paul Kellet

Notes :
(Quoted from Paul's mail)
Below is another waveform generation method based on a train of sinc functions (actually an alternating loop along a sinc between t=0 and t=period/2).

The code integrates the pulse train with a dc offset to get a sawtooth, but other shapes can be made in the usual ways... Note that 'dc' and 'leak' may need to be adjusted for very high or low frequencies.

I don't know how original it is (I ought to read more) but it is of usable quality, particularly at low frequencies. There's some scope for optimisation by using a table for sinc, or maybe a a truncated/windowed sinc?

I think it should be possible to minimise the aliasing by fine tuning 'dp' to slightly less than 1 so the sincs join together neatly, but I haven't found the best way to do it. Any comments gratefully received.

Code :
```
float p=0.0f;      //current position
float dp=1.0f;     //change in postion per sample
float pmax;        //maximum position
float x;           //position in sinc function
float leak=0.995f; //leaky integrator
float dc;          //dc offset
float saw;         //output


//set frequency...

  pmax = 0.5f * getSampleRate() / freqHz;
  dc = -0.498f/pmax;


//for each sample...

  p += dp;
  if(p < 0.0f)
  {
    p = -p;
    dp = -dp;
  }
  else if(p > pmax)
  {
    p = pmax + pmax - p;
    dp = -dp;
  }

  x= pi * p;
  if(x < 0.00001f)
     x=0.00001f; //don't divide by 0

  saw = leak*saw + dc + (float)sin(x)/(x);
```

Cubic polynomial envelopes (click this to go back to the index)

Type : envellope generation

References : Posted by Andy Mucho

Notes :
This function runs from:
startlevel at Time=0
midlevel at Time/2
endlevel at Time
At moments of extreme change over small time, the function can generate out of range (of the 3 input level) numbers, but isn't really a problem in actual use with real numbers, and sensible/real times..

Code :
```
time = 32
startlevel = 0
midlevel = 100
endlevel = 120
k = startlevel + endlevel - (midlevel * 2)
r = startlevel
s = (endlevel - startlevel - (2 * k)) / time
t = (2 * k) / (time * time)
bigr = r
bigs = s + t
bigt = 2 * t

for(int i=0;i<time;i++)
{
 bigr = bigr + bigs
 bigs = bigs + bigt
}
```

Comments
from : thecourier@infinito.it
comment : I have try this and it works fine, but what hell is bigs?????

bye bye

```
        float time = (float)pRect.Width();    //time in sampleframes
 float startlevel = (float)pRect.Height();  //max h vedi ma 1.0
 float midlevel = 500.f;
 float endlevel = 0.f;

 float k = startlevel + endlevel - (midlevel * 2);
 float r = startlevel;
 float s = (endlevel - startlevel - (2 * k)) / time;
 float t= (2 * k) / (time * time);

 float bigr = r;
 float bigs = s + t;
 float bigt = 2 * t;


 for(int i=0;i<time;i++)
 {
  bigr = bigr + bigs;
  bigs = bigs + bigt;      //bigs? co'è
  pDC->SetPixel(i,(int)bigr,RGB (0, 0, 0));
 }
```

Discrete Summation Formula (DSF) (click this to go back to the index)

References : Stylson, Smith and others... (posted by Alexander Kritov)

Notes :
Buzz uses this type of synth.
For cool sounds try to use variable,
for example a=exp(-x/12000)*0.8 // x- num.samples

Code :
```
double DSF (double x,  // input
            double a,  // a<1.0
            double N,  // N<SmplFQ/2,
            double fi) // phase
{
  double s1 = pow(a,N-1.0)*sin((N-1.0)*x+fi);
  double s2 = pow(a,N)*sin(N*x+fi);
  double s3 = a*sin(x+fi);
  double s4 =1.0 - (2*a*cos(x)) +(a*a);
  if (s4==0)
     return 0;
  else
     return (sin(fi) - s3 - s2 +s1)/s4;
}
```

Comments
from : dfl[*AT*]ccrma.stanford.edu
comment : According to Stilson + Smith, this should be

```
double s1 = pow(a,N+1.0)*sin((N-1.0)*x+fi);
                     ^
                     !
```

Could be a typo though?

from : Alex
comment : yepp..

from : TT
comment : So what is wrong about "double" up there?
For DSF, do we have to update the phase (fi input) at every sample?
Another question is what's the input x supposed to represent? Thanks!

from : David Lowenfels
comment : input x should be the phase, and fi is the initial phase I guess? Seems redundant to me.
There is nothing wrong with the double, there is a sign typo in the original posting.

DSF (super-set of BLIT) (click this to go back to the index)

Type : matlab code

References : Posted by David Lowenfels

Notes :
Discrete Summation Formula ala Moorer

computes equivalent to sum{k=0:N-1}(a^k * sin(beta + k*theta))
modified from Emanuel Landeholm's C code
output should never clip past [-1,1]

If using for BLIT synthesis for virtual analog:
N = maxN;
a = attn_at_Nyquist ^ (1/maxN); %hide top harmonic popping in and out when sweeping frequency
beta = pi/2;
num = 1 - a^N * cos(N*theta) - a*(cos(theta) - a^N * cos(N*theta - theta)); %don't waste time on beta

You can also get growing harmonics if a > 1, but the min statement in the code must be removed, and the scaling will be weird.

Code :
```
function output = dsf( freq, a, H, samples, beta)
%a = rolloff coeffecient
%H = number of harmonic overtones (fundamental not included)
%beta = harmonic phase shift

samplerate = 44.1e3;
freq = freq/samplerate; %normalize frequency

bandlimit = samplerate / 2; %Nyquist
maxN = 1 + floor( bandlimit / freq ); %prevent aliasing
N = min(H+2,maxN);

theta = 2*pi * phasor(freq, samples);

epsilon = 1e-6;
a = min(a, 1-epsilon); %prevent divide by zero

num = sin(beta) - a*sin(beta-theta) - a^N*sin(beta + N*theta) + a^(N+1)*sin(beta+(N-1)*theta);
den = (1 + a * ( a - 2*cos(theta) ));

output = 2*(num ./ den - 1) * freq; %subtract by one to remove DC, scale by freq to normalize
output = output * maxN/N;           %OPTIONAL: rescale to give louder output as rolloff increases

function out = phasor(normfreq, samples);
out = mod( (0:samples-1)*normfreq , 1);
out = out * 2 - 1;                  %make bipolar
```

Comments
from : David Lowenfels
comment : oops, there's an error in this version. frequency should not be normalized until after the maxN calculation is done.

Fast LFO in Delphi... (click this to go back to the index)

References : Posted by Dambrin Didier (gol [AT] e-officedirect [DOT] com)

Linked file : LFOGenerator.zip

Notes :
[from Didier's mail...]
[see attached zip file too!]

I was working on a flanger, & needed an LFO for it. I first used a Sin(), but it was too slow, then tried a big wavetable, but it wasn't accurate enough.

I then checked the alternate sine generators from your web site, & while they're good, they all can drift, so you're also wasting too much CPU in branching for the drift checks.

So I made a quick & easy linear LFO, then a sine-like version of it. Can be useful for LFO's, not to output as sound.
If has no branching & is rather simple. 2 Abs() but apparently they're fast. In all cases faster than a Sin()

It's in delphi, but if you understand it you can translate it if you want.
It uses a 32bit integer counter that overflows, & a power for the sine output.
If you don't know delphi, $ is for hex (h at the end in c++?), Single is 32bit float, integer is 32bit integer (signed, normally).

Code :
```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, ComCtrls;

type
  TForm1 = class(TForm)
    PaintBox1: TPaintBox;
    Bevel1: TBevel;
    procedure PaintBox1Paint(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.PaintBox1Paint(Sender: TObject);
var n,Pos,Speed:Integer;
    Output,Scale,HalfScale,PosMul:Single;
    OurSpeed,OurScale:Single;
begin
OurSpeed:=100;  // 100 samples per cycle
OurScale:=100;  // output in -100..100

Pos:=0;  // position in our linear LFO
Speed:=Round($100000000/OurSpeed);


// --- triangle LFO ---
Scale:=OurScale*2;
PosMul:=Scale/$80000000;

// loop
for n:=0 to 299 do
  Begin
  // inc our 32bit integer LFO pos & let it overflow. It will be seen as signed when read by the math unit
  Pos:=Pos+Speed;

  Output:=Abs(Pos*PosMul)-OurScale;

  // visual
  Paintbox1.Canvas.Pixels[n,Round(100+Output)]:=clRed;
  End;


// --- sine-like LFO ---
Scale:=Sqrt(OurScale*4);
PosMul:=Scale/$80000000;
HalfScale:=Scale/2;

// loop
for n:=0 to 299 do
  Begin
```

```
  // inc our 32bit integer LFO pos & let it overflow. It will be seen as signed when read by the math unit
  Pos:=Pos+Speed;

  Output:=Abs(Pos*PosMul)-HalfScale;
  Output:=Output*(Scale-Abs(Output));

  // visual
  Paintbox1.Canvas.Pixels[n,Round(100+Output)]:=clBlue;
  End;
end;

end.
```

Fast sine and cosine calculation (click this to go back to the index)

Type : waveform generation

References : Lot's or references... Check Julius O. SMith mainly

Code :
```
init:
float a = 2.f*(float)sin(Pi*frequency/samplerate);

float s[2];

s[0] = 0.5f;
s[1] = 0.f;

loop:
s[0] = s[0] - a*s[1];
s[1] = s[1] + a*s[0];
output_sine = s[0];
output_cosine = s[1]
```

Comments

from : nigel
comment : This is the Chamberlin state variable filter specialized for infinite Q oscillation. A few things to note:

Like the state variable filter, the upper frequency limit for stability is around one-sixth the sample rate.

The waveform symmetry is very pure at low frequencies, but gets skewed as you get near the upper limit.

For low frequencies, sin(n) is very close to n, so the calculation for "a" can be reduced to a = 2*Pi*frequency/samplerate.

You shouldn't need to resync the oscillator--for fixed point and IEEE floating point, errors cancel exactly, so the osciallator runs forever without drifting in amplitude or frequency.

from : DFL
comment : Yeah, this is a cool trick! :)

FYI you can set s[0] to whatever amplitude of sinewave you desire. With 0.5, you will get +/- 0.5

from : bigtick@pastnotecut.org
comment : After a while it may drift, so you should resync it as follows:
const float tmp=1.5f-0.5f*(s[1]*s[1]+s[0]*s[0]);
s[0]*=tmp; s[1]*=tmp;

This assumes you set s[0] to 1.0 initially.

'Tick

from : DFL
comment : Just to expalin the above "resync" equation
(3-x)/2 is an approximation of 1/sqrt(x)
So the above is actually renormalizing the complex magnitude.
[sin^2 (x) + cos^2(x) = 1]

from : antiprosynthesis@hotmail.com
comment : I made a nice little console 'game' using your cordic sinewave approximation. Download it at http://users.pandora.be/antipro/Other/Ascillator.zip (includes source code). Just for oldschool fun :).

Fast sine wave calculation (click this to go back to the index)

Type : waveform generation

References : James McCartney in Computer Music Journal, also the Julius O. Smith paper

Notes :
(posted by Niels Gorisse)
If you change the frequency, the amplitude rises (pitch lower) or lowers (pitch rise) a LOT I fixed the first problem by thinking about what actually goes wrong. The answer was to recalculate the phase for that frequency and the last value, and then continue normally.

Code :
```
Variables:
ip = phase of the first output sample in radians
w = freq*pi / samplerate
b1 = 2.0 * cos(w)

Init:
y1=sin(ip-w)
y2=sin(ip-2*w)

Loop:
y0 = b1*y1 - y2
y2 = y1
y1 = y0

output is in y0 (y0 = sin(ip + n*freq*pi / samplerate), n= 0, 1, 2, ... I *think*)

Later note by James McCartney:
if you unroll such a loop by 3 you can even eliminate the assigns!!

y0 = b1*y1 - y2
y2 = b1*y0 - y1
y1 = b1*y2 - y0
```

Comments
from : kainhart[at]hotmail.com
comment : try using this to make sine waves with frequency less that 1. I did and it gives very rough half triangle-like waves. Is there any way to fix this? I want to use a sine generated for LFO so I need one that works for low frequencies.

Fast square wave generator (click this to go back to the index)

Type : NON-bandlimited osc...

References : Posted by Wolfgang (wschneider[AT]nexoft.de)

Notes :
Produces a square wave -1.0f .. +1.0f.
The resulting waveform is NOT band-limited, so it's propably of not much use for syntheis. It's rather useful for LFOs and the like, though.

Code :
```
Idea: use integer overflow to avoid conditional jumps.

// init:
typedef unsigned long ui32;

float sampleRate = 44100.0f; // whatever
float freq = 440.0f; // 440 Hz
float one = 1.0f;
ui32 intOver = 0L;
ui32 intIncr = (ui32)(4294967296.0 / hostSampleRate / freq));

// loop:
(*((ui32 *)&one)) &= 0x7FFFFFFF; // mask out sign bit
(*((ui32 *)&one)) |= (intOver & 0x80000000);
intOver += intIncr;
```

Comments
from : musicdsp at lancej.com
comment : So, how would I get the output into a float variable like square_out, for instance?

Gaussian White noise (click this to go back to the index)

References : Posted by Alexey Menshikov

Notes :
Code I use sometimes, but don't remember where I ripped it from.

 - Alexey Menshikov

Code :
```
#define ranf() ((float) rand() / (float) RAND_MAX)

float ranfGauss (int m, float s)
{
   static int pass = 0;
   static float y2;
   float x1, x2, w, y1;

   if (pass)
   {
      y1 = y2;
   } else  {
      do {
         x1 = 2.0f * ranf () - 1.0f;
         x2 = 2.0f * ranf () - 1.0f;
         w = x1 * x1 + x2 * x2;
      } while (w >= 1.0f);

      w = (float)sqrt (-2.0 * log (w) / w);
      y1 = x1 * w;
      y2 = x2 * w;
   }
   pass = !pass;

   return ( (y1 * s + (float) m));
}
```

Comments
from : davidchristenATgmxDOTnet
comment : White Noise does !not! consist of uniformly distributed values. Because in white noise, the power of the frequencies are uniformly distributed. The values must be normal (or gaussian) distributed. This is achieved by the Box-Muller Transformation. This function is the polar form of the Box-Muller Transformation. It is faster and numeriacally more stable than the basic form. The basic form is coded in the other (second) post.
Detailed information on this topic:
http://www.taygeta.com/random/gaussian.html
http://www.eece.unm.edu/faculty/bsanthan/EECE-541/white2.pdf

Cheers David

Gaussian White Noise (click this to go back to the index)

References : Posted by remage[AT]netposta.hu

Notes :
SOURCE:

Steven W. Smith:
The Scientist and Engineer's Guide to Digital Signal Processing
http://www.dspguide.com

Code :
```
#define PI 3.1415926536f

float R1 = (float) rand() / (float) RAND_MAX;
float R2 = (float) rand() / (float) RAND_MAX;

float X = (float) sqrt( -2.0f * log( R1 )) * cos( 2.0f * PI * R2 );
```

Comments
from : pan[at]spinningkids.org
comment : The previous one seems better for me, since it requires only a rand, half log and half sqrt per sample.
Actually, I used that one, but I can't remember where I found it, too. Maybe on Knuth's book.

Inverted parabolic envelope (click this to go back to the index)

Type : envellope generation

References : Posted by James McCartney

Code :
```
dur = duration in samples
midlevel = amplitude at midpoint
beglevel = beginning and ending level (typically zero)

amp = midlevel - beglevel;

rdur = 1.0 / dur;
rdur2 = rdur * rdur;

level = beglevel;
slope = 4.0 * amp * (rdur - rdur2);
curve = -8.0 * amp * rdur2;

...

for (i=0; i<dur; ++i) {
        level += slope;
        slope += curve;
}
```

Comments

from : krakengore@libero.it

comment : This parabola approximation seems more like a linear than a parab/expo envelope... or i'm mistaking something but i tryed everything and is only linear.

from : ex0r0x0r@hotmail.com

comment : slope is linear, but 'slope' is a function of 'curve'. If you imagine you threw a ball upwards, think of 'curve' as the gravity, 'slope' as the vertical velocity, and 'level' as the vertical displacement.

Phase modulation Vs. Frequency modulation (click this to go back to the index)

References : Posted by Bram

Linked file : SimpleOscillator.h (this linked file is included below)

Notes :
This code shows what the difference is betwee FM and PM.
The code is NOT optimised, nor should it be used like this.
It is an EXAMPLE

See linked file.

Linked files

```
////////////////////////////////////////////////////
//
// this code was NEVER MEANT TO BE USED.
//
// use as EXPLANATION ONLY for the difference between
// Phase Modulation and Frequency Modulation.
// there are MANY ways to speed this code up.
//
// bram@musicdsp.org | bram@smartelectronix.com
//
// ps:
// we use the 'previous' value of the phase in all the algo's to make sure that
// the first call to the getSampleXX() function returns the wave at phase 'zero'
//
////////////////////////////////////////////////////

#include "math.h";

#define Pi 3.141592f

class SimpleOscillator
{
 SimpleOscillator(const float sampleRate = 44100.f, const long tableSize = 4096)
 {
  this->tableSize = tableSize;
  this->sampleRate = sampleRate;

  phase = 0.f;

  makeTable();
 }

 ~SimpleOscillator()
 {
  delete [] table;
 }


 // normal oscillator, no modulation
 //
 float generateSample(const float frequency)
 {
  float lookupPhase = phase;

  phase += frequency * (float)tableSize / sampleRate;
  wrap(phase);

  return lookup(lookupPhase);
 }


 // frequency modulation
 // the fm input should be in HZ.
 //
 // example:
 // osc1.getSampleFM(440.f, osc2.getSample(0.5f) * 5.f )
 // would give a signal where the frequency of the signal is
 // modulated between 435hz and 445hz at a 0.5hz rate
 //
 float generateSampleFM(const float frequency, const float fm)
 {
  float lookupPhase = phase;

  phase += (frequency + fm) * (float)tableSize / sampleRate;
  wrap(phase);

  return lookup(lookupPhase);
 }
```

```
 // phase modulation
 //
 // a phase mod value of 1.f will increase the "phase" of the wave by a full cycle
 // i.e. calling getSamplePM(440.f,1.f) will result in the "same" wave as getSamplePM(440.f,0.f)
 //
 float generateSamplePM(const float frequency, const float pm)
 {
  float lookupPhase = phase + (float)tableSize * pm;
  wrap(lookupPhase)

  phase += frequency * (float)tableSize / sampleRate;
  wrap(phase);

  return lookup(lookupPhase);
 }


 // do the lookup in the table
 // you could use different methods here
 // like linear interpollation or higher order...
 // see musicdsp.org
 //
 float lookup(const float phase)
 {
  return table[(long)phase];
 }


 // wrap around
 //
 void wrap(float &in)
 {
  while(in < 0.f)
   in += (float)tableSize;

  while(in >= (float)tableSize)
   in -= (float)tableSize;

  return in;
 }


 // set the sample rate
 //
 void setSampleRate(const float sampleRate)
 {
  this->sampleRate = sampleRate;
 }


 // sets the phase of the oscillator
 // phase should probably be in 0..Pi*2
 //
 void setPhase(const float phase)
 {
  this->phase = phase / (2.f * Pi) * (float)tableSize;
  wrap(phase);
 }

private:

 float sampleRate;
 float phase;

 float *table;
 long tableSize;

 void makeTable()
 {
  table = new float[tableSize];
  for(long i=0;i<tableSize;i++)
  {
   float x = Pi * 2.f * (float)i / (float)tableSize;
   table[i] = (float)sin(x);
  }
 }
}
```

Phase modulation Vs. Frequency modulation II (click this to go back to the index)

References : Posted by James McCartney

Notes :
The difference between FM & PM in a digital oscillator is that FM is added to the frequency before the phase integration, while PM is added to the phase after the phase integration. Phase integration is when the old phase for the oscillator is added to the current frequency (in radians per sample) to get the new phase for the oscillator. The equivalent PM modulator to obtain the same waveform as FM is the integral of the FM modulator. Since the integral of sine waves are inverted cosine waves this is no problem. In modulators with multiple partials, the equivalent PM modulator will have different relative partial amplitudes. For example, the integral of a square wave is a triangle wave; they have the same harmonic content, but the relative partial amplitudes are different. These differences make no difference since we are not trying to exactly recreate FM, but real (or nonreal) instruments.

The reason PM is better is because in PM and FM there can be non-zero energy produced at 0 Hz, which in FM will produce a shift in pitch if the FM wave is used again as a modulator, however in PM the DC component will only produce a phase shift. Another reason PM is better is that the modulation index (which determines the number of sidebands produced and which in normal FM is calculated as the modulator amplitude divided by frequency of modulator) is not dependant on the frequency of the modulator, it is always equal to the amplitude of the modulator in radians. The benefit of solving the DC frequency shift problem, is that cascaded carrier-modulator pairs and feedback modulation are possible. The simpler calculation of modulation index makes it easier to have voices keep the same harmonic structure throughout all pitches.

The basic mathematics of phase modulation are available in any text on electronic communication theory.

Below is some C code for a digital oscillator that implements FM,PM,and AM. It illustrates the difference in implementation of FM & PM. It is only meant as an example, and not as an efficient implementation.

Code :
```
/* Example implementation of digital oscillator with FM, PM, & AM */

#define PI 3.14159265358979
#define RADIANS_TO_INDEX (512.0 / (2.0 * PI))

typedef struct{ /* oscillator data */
 double freq;   /* oscillator frequency in radians per sample */
 double phase;  /* accumulated oscillator phase in radians */
 double wavetable[512]; /* waveform lookup table */
} OscilRec;


/* oscil - compute 1 sample of oscillator output whose freq. phase and
*    wavetable are in the OscilRec structure pointed to by orec.
*/
double oscil(orec, fm, pm, am)
 OscilRec *orec;  /* pointer to the oscil's data */
 double fm; /* frequency modulation input  in radians per sample */
 double pm; /* phase modulation input      in radians */
 double am; /* amplitude modulation input  in any units you want */
{
 long tableindex;            /* index into wavetable */
 double instantaneous_freq;  /* oscillator freq  + freq  modulation */
 double instantaneous_phase; /* oscillator phase + phase modulation */
 double output;              /* oscillator output */

 instantaneous_freq  = orec->freq  + fm; /* get instantaneous freq */
 orec->phase += instantaneous_freq;      /* accumulate phase */
 instantaneous_phase = orec->phase + pm; /* get instantaneous phase */

 /* convert to lookup table index */
 tableindex = RADIANS_TO_INDEX * instantaneous_phase;
 tableindex &= 511; /* make it mod 512 === eliminate multiples of 2*k*PI */

 output = orec->wavetable[tableindex] * am; /* lookup and mult by am input */

 return (output);  /* return oscillator output */
}
```

Pseudo-Random generator (click this to go back to the index)

Type : Linear Congruential, 32bit

References : Hal Chamberlain, "Musical Applications of Microprocessors" (Posted by Phil Burk)

Notes :
This can be used to generate random numeric sequences or to synthesise a white noise audio signal.
If you only use some of the bits, use the most significant bits by shifting right.
Do not just mask off the low bits.

Code :
```
/* Calculate pseudo-random 32 bit number based on linear congruential method. */
unsigned long GenerateRandomNumber( void )
{
   /* Change this for different random sequences. */
   static unsigned long randSeed = 22222;
   randSeed = (randSeed * 196314165) + 907633515;
   return randSeed;
}
```

Pulsewidth modulation (click this to go back to the index)

Type : waveform generation

References : Steffan Diedrichsen

Notes :
Take an upramping sawtooth and its inverse, a downramping sawtooth. Adding these two waves with a well defined delay between 0 and period (1/f) results in a square wave with a duty cycle ranging from 0 to 100%.

quick and dirty sine generator (click this to go back to the index)

Type : sine generator

References : Posted by couriervst[AT]hotmail[DOT]com

Notes :
this is part of my library, although I've seen a lot of sine generators, I've never seen the simplest one, so I try to do it,
tell me something, I've try it and work so tell me something about it

Code :
```
PSPsample PSPsin1::doOsc(int numCh)
{

 double x=0;
 double t=0;

 if(m_time[numCh]>m_sampleRate) //re-init cycle
  m_time[numCh]=0;

 if(m_time[numCh]>0)
 {
  t =(double)(((double)m_time[numCh])/(double)m_sampleRate);

  x=(m_2PI *(double)(t)*m_freq);
 }
 else
  x=0;


 PSPsample r=(PSPsample) sin(x+m_phase)*m_amp;

 m_time[numCh]++;

 return  r;

}
```

Comments
from : pete@bannister25.plus.com
comment : isn't the sin() function a little bit heavyweight? Since this is based upon slices of time, would it not be much more processor efficient to use a state variable filter that is self oscillating?

The operation:
t =(double)(((double)m_time[numCh])/(double)m_sampleRate);

also seems a little bit much, since t could be calculated by adding an interval value, which would eliminate the divide (needs more clocks). The divide would then only need to be done once.
An FDIV may take 39 clock cycles minimum(depending on the operands), whilst an FADD is far faster (3 clocks). An FMUL is comparable to an add, which would be a predominant instruction if using the SVF method.

FSIN may take between 16-126 clock cylces.

(clock cycle info nabbed from: http://www.singlix.com/trdos/pentium.txt)

from : rossb@audiomulch.com
comment : See also the fun with sinusoids page:
http://www.audiomulch.com/~rossb/code/sinusoids/

SawSin (click this to go back to the index)

Type : Oscillator shape

References : Posted by Alexander Kritov

Code :
```
double sawsin(double x)
{
   double t = fmod(x/(2*M_PI),(double)1.0);
   if (t>0.5)
       return -sin(x);
   if (t<=0.5)
       return (double)2.0*t-1.0;
}
```

Sine calculation (click this to go back to the index)

Type : waveform generation, Taylor approximation of sin()

References : Posted by Phil Burk

Notes :
Code from JSyn for a sine wave generator based on a Taylor Expansion. It is not as efficient as the filter methods, but it has linear frequency control and is, therefore, suitable for FM or other time varying applications where accurate frequency is needed. The sine generated is accurate to at least 16 bits.

Code :
```
for(i=0; i < nSamples ; i++)
{
  //Generate sawtooth phasor to provide phase for sine generation
  IncrementWrapPhase(phase, freqPtr[i]);
  //Wrap phase back into region where results are more accurate

  if(phase > 0.5)
    yp = 1.0 - phase;
  else
  {
    if(phase < -0.5)
       yp = -1.0 - phase;
    else
        yp = phase;
  }

  x = yp * PI;
  x2 = x*x;

  //Taylor expansion out to x**9/9! factored  into multiply-adds
  fastsin = x*(x2*(x2*(x2*(x2*(1.0/362880.0)
            - (1.0/5040.0))
            + (1.0/120.0))
            - (1.0/6.0))
            + 1.0);

  outPtr[i] = fastsin * amplPtr[i];
}
```

Square Waves (click this to go back to the index)

Type : waveform generation

References : Posted by Sean Costello

Notes :
One way to do a square wave:

You need two buzz generators (see Dodge & Jerse, or the Csound source code, for implementation details). One of the buzz generators runs at the desired square wave frequency, while the second buzz generator is exactly one octave above this pitch. Subtract the higher octave buzz generator's output from the lower buzz generator's output - the result should be a signal with all odd harmonics, all at equal amplitude. Filter the resultant signal (maybe integrate it). Voila, a bandlimited square wave! Well, I think it should work...

The one question I have with the above technique is whether it produces a waveform that truly resembles a square wave in the time domain. Even if the number of harmonics, and the relative ratio of the harmonics, is identical to an "ideal" bandwidth-limited square wave, it may have an entirely different waveshape. No big deal, unless the signal is processed by a nonlinearity, in which case the results of the nonlinear processing will be far different than the processing of a waveform that has a similar shape to a square wave.

Comments
from : dfl AT stanford. edu
comment : Actually, I don't think this would work...
The proper way to do it is subtract a phase shifted buzz (aka BLIT) at the same frequency. This is equivalent to comb filtering, which will notch out the even harmonics.

Waveform generator using MinBLEPS (click this to go back to the index)

References : Posted by locke[AT]rpgfan.demon.co.uk

Linked file : MinBLEPS.zip

Notes :
C code and project file for MSVC6 for a bandwidth-limited saw/square (with PWM) generator using MinBLEPS.

This code is based on Eli's MATLAB MinBLEP code and uses his original minblep.mat file.
Instead of keeping a list of all active MinBLEPS, the output of each MinBLEP is stored in a buffer, in which all consequent MinBLEPS and the waveform output are added together. This optimization makes it fast enough to be used realtime.

Produces slight aliasing when sweeping high frequencies. I don't know wether Eli's original code does the same, because I don't have MATLAB. Any help would be appreciated.

The project name is 'hardsync', because it's easy to generate hardsync using MinBLEPS.

Code :

Wavetable Synthesis (click this to go back to the index)

References : Robert Bristow-Johnson

Linked file : http://www.harmony-central.com/Synth/Articles/Wavetable_101/Wavetable-101.pdf

Notes :
Wavetable sythesis AES paper by RBJ.

Weird synthesis (click this to go back to the index)

References : Posted by Andy M00cho

Notes :
(quoted from Andy's mail...)
What I've done in a soft-synth I've been working on is used what I've termed Fooglers, no reason, just liked the name :) Anyway all I've done is use a *VERY* short delay line of 256 samples and then use 2 controllable taps into the delay with High Frequency Damping, and a feedback parameter.

Using a tiny fixed delay size of approx. 4.8ms (really 256 samples/1k memory with floats) means this costs, in terms of cpu consumption practically nothing, and the filter is a real simple 1 pole low-pass filter. Maybe not DSP'litically correct but all I wanted was to avoid the high frequencies trashing the delay line when high feedbacks (99%->99.9%) are used (when the fun starts ;).

I've been getting some really sexy sounds out of this idea, and of course you can have the delay line tuneable if you choose to use fractional taps, but I'm happy with it as it is.. 1 nice simple, yet powerful addition to the base oscillators.

In reality you don't need 2 taps, but I found that using 2 added that extra element of funkiness...

Comments
from : philmagnotta@aol.com
comment : Andy:
I'm curious about your delay line. It's length is
4.8 m.sec.fixed. What are the variables in the two controllable taps and is the 6dB filter variable frequency wise?
Phil

from : electropop@yahoo.com
comment : What you have there is the core of a physical modelling algorithm. I have done virtually the same thing to model plucked string instruments in Reaktor. It's amazingly realistic. See http://joeorgren.com

Coefficients for Daubechies wavelets 1-38 (click this to go back to the index)

Type : wavelet transform

References : Computed by Kazuo Hatano, Compiled and verified by Olli Niemitalo

Linked file : daub.h (this linked file is included below)

Linked files

```
/* Coefficients for Daubechies wavelets 1-38
 * -----------------------------------------
 * 2000.08.10 - Some more info added.
 *
 * Computed by Kazuo Hatano, Aichi Institute of Technology.
 * ftp://phase.etl.go.jp/pub/phase/wavelet/index.html
 *
 * Compiled and verified by Olli Niemitalo.
 *
 * Discrete Wavelet Transformation (DWT) breaks a signal down into
 * subbands distributed logarithimically in frequency, each sampled
 * at a rate that has a natural proportion to the frequencies in that
 * band. The traditional fourier transformation has no time domain
 * resolution at all, or when done using many short windows on a
 * longer data, equal resolution at all frequencies. The distribution
 * of samples in the time and frequency domain by DWT is of form:
 *
 * log f
 *  |XXXXXXXXXXXXXXXX  X = a sample
 *  |X X X X X X X X   f = frequency
 *  |X   X   X   X     t = time
 *  |X       X
 *  |X
 *   ----------------t
 *
 * Single
 * subband decomposition         and      reconstruction:
 *
 *     -> high -> decimate -------------> dilute -> high
 *    |   pass    by 2      high subband  by 2      pass \
 * in |                                                   + out
 *    |                                                  /  =in
 *     -> low  -> decimate -------------> dilute -> low
 *        pass    by 2      low subband   by 2      pass
 *
 * This creates two subbands from the input signal, both sampled at half
 * the original frequency. The filters approximate halfband FIR filters
 * and are determined by the choice of wavelet. Using Daubechies wavelets
 * (and most others), the data can be reconstructed to the exact original
 * even when the halfband filters are not perfect. Note that the amount
 * of information (samples) stays the same throughout the operation.
 *
 * Decimation by 2: ABCDEFGHIJKLMNOPQR -> ACEGIKMOQ
 *   Dilution by 2: ACEGIKMOQ -> A0C0E0G0I0K0M0O0Q0
 *
 * To get the logarithmic resolution in frequency, the low subband is
 * re-transformed, and again, the low subband from this transformation
 * gets the same treatment etc.
 *
 * Decomposition:
 *
 *     -> high -> decimate --------------------------------> subband0
 *    |   pass    by 2
 * in |                 -> high -> decimate ---------------> subband1
 *    |                |   pass    by 2
 *     -> low -> decim |                 -> high -> decim -> subband2
 *        pass   by 2  |                |   pass     by 2
 *                      -> low -> decim |
 *                         pass   by 2  |   .   down to what suffices
 *                                       -> .    or if periodic data,
 *                                          .     until short of data
 * Reconstruction:
 *
 * subband0 -----------------------------------> dilute -> high
 *                                               by 2      pass \
 * subband1 ------------------> dilute -> high                   + out
 *                              by 2      pass \                /  =in
 * subband2 -> dilute -> high                   + dilute -> low
 *             by 2      pass \                /   by 2     pass
 *                             + dilute -> low
 * Start   .                  /   by 2     pass
 * here!   . -> dilute -> low
```

```
 *         .    by 2      pass
 *
 * In a real-time application, the filters introduce delays, so you need
 * to compensate them by adding additional delays to less-delayed higher
 * bands, to get the summation work as intended.
 *
 * For periodic signals or windowed operation, this problem doesn't exist -
 * a single subband transformation is a matrix multiplication, with wrapping
 * implemented in the matrix:
 *
 * Decomposition:
 *
 * |L0|   |C0  C1  C2  C3                | |I0|     L = lowpass output
 * |H0|   |C3 -C2  C1 -C0                | |I1|     H = highpass output
 * |L1|   |        C0  C1  C2  C3        | |I2|     I = input
 * |H1| = |        C3 -C2  C1 -C0        | |I3|     C = coefficients
 * |L2|   |                C0  C1  C2  C3| |I4|
 * |H2|   |                C3 -C2  C1 -C0| |I5|
 * |L3|   |C2  C3                  C0  C1| |I6|
 * |H3|   |C1 -C0                  C3 -C2| |I7|     Daubechies 4-coef:
 *
 *      1+sqrt(3)        3+sqrt(3)        3-sqrt(3)        1-sqrt(3)
 * C0 = ---------   C1 = ---------   C2 = ---------   C3 = ---------
 *      4 sqrt(2)        4 sqrt(2)        4 sqrt(2)        4 sqrt(2)
 *
 * Reconstruction:
 *
 * |I0|   |C0  C3                  C2  C1| |L0|
 * |I1|   |C1 -C2                  C3 -C0| |H0|
 * |I2|   |C2  C1  C0  C3                | |L1|
 * |I3| = |C3 -C0  C1 -C2                | |H1|
 * |I4|   |        C2  C1  C0  C3        | |L2|
 * |I5|   |        C3 -C0  C1 -C2        | |H2|
 * |I6|   |                C2  C1  C0  C3| |L3|
 * |I7|   |                C3 -C0  C1 -C2| |H3|
 *
 * This file contains the lowpass FIR filter coefficients. Highpass
 * coefficients you get by reversing tap order and multiplying by
 * sequence 1,-1, 1,-1, ... Because these are orthogonal wavelets, the
 * analysis and reconstruction coefficients are the same.
 *
 * A coefficient set convolved by its reverse is an ideal halfband lowpass
 * filter multiplied by a symmetric windowing function. This creates the
 * kind of symmetry in the frequency domain that enables aliasing-free
 * reconstruction. Daubechies wavelets are the minimum-phase, minimum
 * number of taps solutions for a number of vanishing moments (seven in
 * Daub7 etc), which determines their frequency selectivity.
 */

const double Daub1[2] = {
 7.071067811865475244008443621048490392848359376884740365883398e-01,
 7.071067811865475244008443621048490392848359376884740365883398e-01};

const double Daub2[4] = {
 4.829629131445341433748715998644486838169524195042022752011715e-01,
 8.365163037378079055752937809168732034593703883484392934953414e-01,
 2.241438680420133810259727622404003554678835181842717613871683e-01,
-1.294095225512603811744494188120241641745344506599652569070016e-01};

const double Daub3[6] = {
 3.326705529500826159985115891390056300129233992450683597084705e-01,
 8.068915093110925764944936040887134905192973949948236181650920e-01,
 4.598775021184915700951519421476167208081101774314923066433867e-01,
-1.350110200102545886963899066993744805622198452237811919756862e-01,
-8.544127388202666169281916918177331153619763898808662976351748e-02,
 3.522629188570953660274066471551002932775838791743161039893406e-02};

const double Daub4[8] = {
 2.303778133088965008632911830440708500016152482483092977910968e-01,
 7.148465705529156470899219552739926037076084010993081758450110e-01,
 6.308807679298589078817163383006152202032229226771951174057473e-01,
-2.798376941685985421141374718007538541198732022449175284003358e-02,
-1.870348117190930840795706727890814195845441743745800912057770e-01,
 3.084138183556076362721936253495905017031482172003403341821219e-02,
 3.288301166688519973540751354924438866454194113754971259727278e-02,
-1.059740178506903210488320852402722918109996490637641983484974e-02};

const double Daub5[10] = {
 1.601023979741929144807237480204207336505441246250578327725699e-01,
 6.038292697971896705401193065250621075074221631016986987969283e-01,
 7.243085284377729277280712441022186407687562182320073725767335e-01,
 1.384281459013207315053971463390246973141057911739561022694652e-01,
-2.422948870663820318625713794746163619914908080626185983913726e-01,
```

```
-3.224486958463837464847975506213492831356498416379847225434268e-02,
 7.757149384004571352313048938860181980623099452012527983210146e-02,
-6.241490212798274274190519112920192970763557165687607323417435e-03,
-1.258075199908199946850973993177579294920459162609785020169232e-02,
 3.335725285473771277998183415817355747636524742305315099706428e-03};

const double Daub6[12] = {
 1.115407433501094636213239172409234390425395919844216759082360e-01,
 4.946238903984530856772041768778555886377863828962743623531834e-01,
 7.511339080210953506789344984397316855802547833382612009730420e-01,
 3.152503517091976290859896548109263966495199235172945244404163e-01,
-2.262646939654398200763145006609034656705401539728969940143487e-01,
-1.297668675672619355622896058765854608452337492235814701599310e-01,
 9.750160558732304910234355253812534233983074749525514279893193e-02,
 2.752286553030572862554083950419321365738758783043454321494202e-02,
-3.158203931748602956507908069984866905747953237314842337511464e-02,
 5.538422011614961392519183980465012206110262773864964295476524e-04,
 4.777257510945510639635975246820707050230501216581434297593254e-03,
-1.077301085308479564852621609587200035235233609334419689818580e-03};

const double Daub7[14] = {
 7.785205408500917901996352195789374837918305292795568438702937e-02,
 3.965393194819173065390003909368428563587151149333287401110499e-01,
 7.291320908462351199169430703392820517179660611901363782697715e-01,
 4.697822874051931224715911609744517386817913056787359532392529e-01,
-1.439060039285649754050683622130460017952735705499084834401753e-01,
-2.240361849938749826381404202332509644757830896773246552665095e-01,
 7.130921926683026475087657050112904822711327451412314659575113e-02,
 8.061260915108307191292248035938190585823820965629489058139218e-02,
-3.802993693501441357959206160185803585446196938467869898283122e-02,
-1.657454163066688065410767489170265479204504394820713705239272e-02,
 1.255099855609984061298988603418777957289474046048710038411818e-02,
 4.295779729213665211321291228197322228235350396942409742946366e-04,
-1.801640704047490915268262912739550962585651469641090625323864e-03,
 3.537137999745202484462958363064254310959060059520040012524275e-04};

const double Daub8[16] = {
 5.441584224310400995500940520299935503599554294733050397729280e-02,
 3.128715909142999706591623755057177219497319740370229185698712e-01,
 6.756307362972898068078007670471831499869115906336364227766759e-01,
 5.853546836542067127712655200450981944303266678053369055707175e-01,
-1.582910525634930566738054787646630415774471154502826559735335e-02,
-2.840155429615469265162031323741647324684350124871451793599204e-01,
 4.724845739132827703605900098258949861948011288770074644084096e-04,
 1.287474266204784588570292875097083843022601575556488795577000e-01,
-1.736930100180754616961614886809598311413086529488394316977315e-02,
-4.408825393079475150676372323896350189751839190110996472750391e-02,
 1.398102791739828164872293057263345144239559532934347169146368e-02,
 8.746094047405776716382743246475640180402147081140676742686747e-03,
-4.870352993451574310422181557109824016634978512157003764736208e-03,
-3.917403733769470462980803573237762675229350073890493724492694e-04,
 6.754494064505693663695475738792991218489630013558432103617077e-04,
-1.174767841247695337306282316988909444086693950311503927620013e-04};

const double Daub9[18] = {
 3.807794736387834658869765887955118448771714496278417476647192e-02,
 2.438346746125903537320415816492844155263611085609231361429088e-01,
 6.048231236901111119030768674342361708959562711896117565333713e-01,
 6.572880780513005380782126390451732140305858669245918854436034e-01,
 1.331973858250075761909549458997955536921780768433661136154346e-01,
-2.932737832791749088064031952421987310438961628589906825725112e-01,
-9.684078322297646051350813353769660224825458104599099679471267e-02,
 1.485407493381063801350727175060423024791258577280603060771649e-01,
 3.072568147933337921231740072037882714105805024670744781503060e-02,
-6.763282906132997367564227482971901592578790871353739900748331e-02,
 2.509471148314519575871897499885543315176271993709633321834164e-04,
 2.236166212367909720537378270269095241855646688308853754721816e-02,
-4.723204757751397277925707848242465405729514912627938018758526e-03,
-4.281503682463429834496795002314531876481181811463288374860455e-03,
 1.847646883056226476619129491125677051121081359600318160732515e-03,
 2.303857635231959672052163928245421692940662052463711972260006e-04,
-2.519631889427101369749886842878606607282181543478028214134265e-04,
 3.934732031627159948068988306589150707782477055517013507359938e-05};

const double Daub10[20] = {
 2.667005790055555358661744877130858277192498290851289932779975e-02,
 1.881768000776914890208929736790939942702546758640393484348595e-01,
 5.272011889317255864817448279595081924981402680840223445318549e-01,
 6.884590394536035657418717825492358539771364042407339537279681e-01,
 2.811723436605774607487269984455892876243888859026150413831543e-01,
-2.498464243273153794161018979207791000564669737132073715013121e-01,
-1.959462743773770435042992543190981318766776476382778474396781e-01,
 1.273693403357932600826772332014009770786177480422245995563097e-01,
```

```
 9.305736460357235116035228983545273226942917998946925868063974e-02,
-7.139414716639708714533609307605064767292611983702150917523756e-02,
-2.945753682187581285828323760141839199388200516064948779769654e-02,
 3.321267405934100173976365318215912897978337413267096043323351e-02,
 3.606553566956169655423291417133403299517350518618994762730612e-03,
-1.073317548333057504431811410651364448111548781143923213370333e-02,
 1.395351747052901165789318447957707567660542855688552426721117e-03,
 1.992405295185056117158742242640643211762555365514105280067936e-03,
-6.858566949597116265613709819265714196625043336786920516211903e-04,
-1.164668551292854509514809710258991891527461854347597362819235e-04,
 9.358867032006959133405013034222854399688456215297276443521873e-05,
-1.326420289452124481243667531226683305749240960605829756400674e-05};

const double Daub11[22] = {
 1.869429776147108402543572939561975728967774455921958543286692e-02,
 1.440670211506245127951915849361001143023718967556239604318852e-01,
 4.498997643560453347688940373853603677806895378648933474599655e-01,
 6.856867749162005111209386316963097935940204964567703495051589e-01,
 4.119643689479074629259396485710667307430400410187845315697242e-01,
-1.622752450274903622405827269985511540744264324212130209649667e-01,
-2.742308468179469612021009452835266628648089521775178221905778e-01,
 6.604358819668319190061457888126302656753142168940791541113457e-02,
 1.498120124663784964066562617044193298588272420267484653796909e-01,
-4.647995511668418727161722589023744577223260966848260747450320e-02,
-6.643878569502520527899215536971203191819566896079739622858574e-02,
 3.133509021904607603094798408303144536358105680880031964936445e-02,
 2.084090436018106302294811255656491015157761832734715691126692e-02,
-1.536482090620159942619811609958822744014326495773000120205848e-02,
-3.340858873014445606090808617982406101930658359499190845656731e-03,
 4.928417656059041123170739741708273690285547729915802418397458e-03,
-3.085928588151431651754590726278953307180216605078488581921562e-04,
-8.930232506662646133900824622648653989879519878620728793133358e-04,
 2.491525235528234988712216872666801088221199302855425381971392e-04,
 5.443907469936847167357856879576832191936678525600793978043688e-05,
-3.463498418698499554128085159974043214506488048233458035943601e-05,
 4.494274277236510095415648282310130916410497987383753460571741e-06};

const double Daub12[24] = {
 1.311225795722951750674609088893328065665510641931325007748280e-02,
 1.095662728211851546057045050248905426075680503066774046383657e-01,
 3.773551352142126570928212604879206149010941706057526334705839e-01,
 6.571987225793070893027611286641169834250203289988412141394281e-01,
 5.158864784278156087560326480543032700677693087036090056127647e-01,
-4.476388565377462666762747311540166529284543631505924139071704e-02,
-3.161784537527855368648029353478031098508839032547364389574203e-01,
-2.377925725606972768399754609133225784553366558331741152482612e-02,
 1.824786059275796798540436116189241710294771448096302698329011e-01,
 5.359569674352150328276276729768332288862665184192705821636342e-03,
-9.643212009650708202650320534322484127430880143045220514346402e-02,
 1.084913025582218438089010237748152188661630567603334659322512e-02,
 4.154627749508444073927094681906574864513532221388374861287078e-02,
-1.221864906974828071998798266471567712982466093116558175344811e-02,
-1.284082519830068329466034471894728496206109832314097633275225e-02,
 6.711499008795509177767027068215672450648112185856456740379455e-03,
 2.248607240995237599950865211267234018343199786146177099262010e-03,
-2.179503618627760471598903379584171187840075291860571264980942e-03,
 6.545128212509595566500430399327110729111770568897356630714552e-06,
 3.886530628209314435897288837795981791917488573420177523436096e-04,
-8.850410920820432420821645961553726598738322151471932808015443e-05,
-2.424154575703078402978915320531719580423778362664282239377532e-05,
 1.277695221937976658714046362616620887375960941439428756055353e-05,
-1.529071758068510902712239164522901223197615439660340672602696e-06};

const double Daub13[26] = {
 9.202133538962367972970163475644184667534171916416562386009703e-03,
 8.286124387290277964432027131230466405208113332890135072514277e-02,
 3.119963221604380633960784112214049693946683528967180317160390e-01,
 6.110558511587876528211995136744180562073612676018239438526582e-01,
 5.888895704312189080710395347395333927665986382812836042235573e-01,
 8.698572617964723731023739838087494399231884076619701250882016e-02,
-3.149729077113886329981698255932282582876888450678789025950306e-01,
-1.245767307508152589413808336021260180792739295173634719572069e-01,
 1.794760794293398432348450072339369013581966256244133393042881e-01,
 7.294893365677716380902830610477661983325929026879873553627963e-02,
-1.058076181879343264509667304196464849478860754801236658232360e-01,
-2.648840647534369463963912248034785726419604844297697016264224e-02,
 5.613947710028342886214501998387331119988378792543100244737056e-02,
 2.379972254059078811465170958554208358094394612051934868475139e-03,
-2.383142071032364903206403067757739134252922717636226274077298e-02,
 3.923941448797416243316370220815526558824746623451404043918407e-03,
 7.255589401617566194518393300502698898973529679646683695269828e-03,
-2.761911234656862178014576266098445995350093330501818024966316e-03,
-1.315673911892298936613835370593643376060412592653652307238124e-03,
```

```
 9.323261308672633862226517802548514100918088299801952307991569e-04,
```

```
-6.239722752474871765674503394120025865444656311678760990761458e-03,
 7.592423604427631582148498743941422461530405946100943351940313e-02,
-7.588974368857737638494890864636995796586975144990925400097160e-03,
-3.688839769173014233352666320894554314718748429706730831064068e-02,
 1.029765964095596941165000580076616900528856265803662208854147e-02,
 1.399376885982873102950451873670329726409840291727868988490100e-02,
-6.990014563413916670284249536517288338057856199646469078115759e-03,
-3.644279621498389932169000540933629387055333973353108668841215e-03,
 3.128023381206268831661202559854678767821471906193608117450360e-03,
 4.078969808497128362417470323406095782431952972310546715071397e-04,
-9.410217493595675889266453953635875407754747216734480509250273e-04,
 1.142415200387223926440228099555662945839684344936472652877091e-04,
 1.747872452253381803801758637660746874986024728615399897971953e-04,
-6.103596621410935835162369150522212811957259981965919143961722e-05,
-1.394566898820889345199078311998401982325273569198675335408707e-05,
 1.133660866127625858758848762886536997519471068203753661757843e-05,
-1.043571342311606501525454737262615404887478930635676471546032e-06,
-7.363656785451205512099695719725563646585445545841663327433569e-07,
 2.308784086857545866405412732942006121306306735866655525372544e-07,
-2.109339630100743097000572623603489906836297584591605307745349e-08};

const double Daub17[34] = {
 2.241807001037312853535962677074436914062191880560370733250531e-03,
 2.598539370360604338914864591720788315473944524878241294399948e-02,
 1.312149033078244065775506231859069960144293609259978530067004e-01,
 3.703507241526411504492548190721886449477078876896803823650425e-01,
 6.109966156846228181886678867679372082737093893358726291371783e-01,
 5.183157640569378393254538528085968046216817197718416402439904e-01,
 2.731497040329363500431250719147586480350469818964563003672942e-02,
-3.283207483639617360909665340725061767581597698151558024679130e-01,
-1.265997522158827028744679110933825505053966260104086162103728e-01,
 1.973105895650109927854047044781930142551422414135646917122284e-01,
 1.011354891774702721509699856433434802196622545499664876109437e-01,
-1.268156917782863110948571128662331680384792185915017065732137e-01,
-5.709141963167692728911239478651382324161160869845347053990144e-02,
 8.110598665416088507965885748555429201024364190954499194020678e-02,
 2.231233617810379595339136059534813756232242114093689244020869e-02,
-4.692243838926973733300897059211400507138768125498030602878439e-02,
-3.270955535819293781655360222177494452069525958061609392809275e-03,
 2.273367658394627031845616244788448969906713741338339498024864e-02,
-3.042989981354637068592482637907206078633395457225096588287881e-03,
-8.602921520322854831713706413243659917926736284271730611920986e-03,
 2.967996691526094872806485060008038269959463846548378995044195e-03,
 2.301205242153545624302059869038423604241976680189447476064764e-03,
-1.436845304802976126222890402980384903503674530729935809561434e-03,
-3.281325194098379713954444017520115075812402442728749700195651e-04,
 4.394654277686436778385677527317841632289249319738892179465910e-04,
-2.561010956654845882729891210949920221664082061531909655178413e-05,
-8.204803202453391839095482576282189866136273049636764338689593e-05,
 2.318681379874595084482068205706277572106695174091895338530734e-05,
 6.990600985076751273204549700855378627762758585902057964027481e-06,
-4.505942477222988194102268206378312129713572600716499944918416e-06,
 3.016549609994557415605207594879939763476168705217646897702706e-07,
 2.957700933316856754979905258816151367870345628924317307354639e-07,
-8.423948446002680178787071296922877068410310942222799622593133e-08,
 7.267492968561608110879767441409035034158581719789791088892046e-09};

const double Daub18[36] = {
 1.576310218440760431540744929939777747670753710991660363684429e-03,
 1.928853172414637705921391715829052419954667025288497572236714e-02,
 1.035884658224235962241910491937253596470696555220241672976224e-01,
 3.146789413370316990571998255652579931786706190489374509491307e-01,
 5.718268077666072234818589370900623419393673743130930561295324e-01,
 5.718016548886513352891119994065965025668047882818525060759395e-01,
 1.472231119699281415750977271081072312557864107355701387801677e-01,
-2.936540407365587442479030994981150723935710729035053239661752e-01,
-2.164809340051429711237678625668271471437937235669492408388692e-01,
 1.495339755653777893509301738913667208804816691893765610261943e-01,
 1.670813127632574045149318139950134745324205646353988083152250e-01,
-9.233188415084628060429372558659459731431848000144569612074508e-02,
-1.067522466598284855932200581614984861385266404624112083917702e-01,
 6.488721621190544281947577955141911463129382116634147846137149e-02,
 5.705124773853688412090768846499622260596226120431038524600676e-02,
-4.452614190298232471556143559744653492971477891439833592755034e-02,
-2.373321039586000103275209582665216110197519330713490233071565e-02,
 2.667070592647059029987908631672020343207895999936072813363471e-02,
 6.262167954305707485236093144497882501990325204745013190268052e-03,
-1.305148094661200177277636447600807169755191054507571666606133e-02,
 1.186300338581174657301741592161819084544899417452317405185615e-04,
 4.943343605466738130665529516802974834299638313366477765295203e-03,
-1.118732666992497072800658855238650182318060482584970145512687e-03,
-1.340596298336106629517567228251583609823044524685986640323942e-03,
 6.284656829651457125619449885420838217551022796301582874349652e-04,
```

```
 2.135815619103406884039052814341926025873200325996466522543440e-04,
-1.986485523117479485798245416362489554927797880264017876139605e-04,
-1.535917123534724675069770335876717193700472427021513236587288e-07,
 3.741237880740038181092208138035393952304292615793985030731363e-05,
-8.520602537446695203919254911655523022437596956226376512305917e-06,
-3.332634478885821888782452033341036827311505907796498439829337e-06,
 1.768712983627615455876328730755375176412501359114058815453100e-06,
-7.691632689885176146000152878539598405817397588156525116769908e-08,
-1.176098767028231698450982356561292561347579777695396953528141e-07,
 3.068835863045174800935478294933975372450179787894574492930570e-08,
-2.507934454948598267195173183147126731806317144868275819941403e-09};

const double Daub19[38] = {
 1.108669763181710571099154195209715164245299677773435932135455e-03,
 1.428109845076439737439889152950199234745663442163665957870715e-02,
 8.127811326545955065296306784901624839844979971028620366497726e-02,
 2.643884317408967846748100380289426873862377807211920718417385e-01,
 5.244363774646549153360575975484064626044633641048072116393160e-01,
 6.017045491275378948867077135921802620536565639585963293313931e-01,
 2.608949526510388292872456675310528324172673101301907739925213e-01,
-2.280913942154826463746325776054637207093787237086425909534822e-01,
-2.858386317558262418545975695028984237217356095588335149922119e-01,
 7.465226970810326636763433111878819005865866149731909656365399e-02,
 2.123497433062784888090608567059824197077074200878839448416908e-01,
-3.351854190230287868169388418785731506977845075238966819814032e-02,
-1.427856950387365749779602731626112812998497706152428508627562e-01,
 2.758435062562866875014743520162198655374474596963423080762818e-02,
 8.690675555581223248847645428808443034785208002468192759640352e-02,
-2.650123625012304089901835843676387361075068017686747808171345e-02,
-4.567422627723090805645444214295796017938935732115630050880109e-02,
 2.162376740958504713032984257172372354318097067858752542571020e-02,
 1.937554988917612764637094354457999814496885095875825546406963e-02,
-1.398838867853514163250401235248662521916813867453095836808366e-02,
-5.866922281012174726584493436054373773814608340808758177372765e-03,
 7.040747367105243153014511207400620109401689897665383078229398e-03,
 7.689543592575483559749139148673955163477947086039406129546422e-04,
-2.687551800701582003957363855070398636534038920982478290170267e-03,
 3.418086534585957765651657290463808135214214848819517257794031e-04,
 7.358025205054352070260481905397281875183175792779904858189494e-04,
-2.606761356786280057318315130897522790383939362073563408613547e-04,
-1.246007917341587753449784408901653990317341413341980904757592e-04,
 8.711270467219922965416862388191128268412933893282083517729443e-05,
 5.105950487073886053049222809934231573687367992106282669389264e-06,
-1.664017629715494454620677719899198630333675608812018108739144e-05,
 3.010964316296526339695334454725943632645798938162427168851382e-06,
 1.531931476691193063931832381086636031203123032723477463624141e-06,
-6.862755657769142701883554613486732854452740752771392411758418e-07,
 1.447088298797844542078219863291615420551673574071367834316167e-08,
 4.636937775782604223430857728210948898871748291085962296649320e-08,
-1.116402067035825816390504769142472586464975799284473682246076e-08,
 8.666848838997619350323013540782124627289742190273059319122840e-10};

const double Daub20[40] = {
 7.799536136668463215861994818889370970510722039232863880031127e-04,
 1.054939462495039832454480973015641498231961468733236691299796e-02,
 6.342378045908151497587346582668785136406523315729666353643372e-02,
 2.199421135513970450080335972537209392121306761010882209298252e-01,
 4.726961853109016963710241465101446230757804141171727845834637e-01,
 6.104932389385938201631515660084201906858628924695448898824748e-01,
 3.615022987393310629195602665268631744967084723079677894136358e-01,
-1.392120880114838725806970545155530518264944915437808314813582e-01,
-3.267868004340349674031122837905370666716645587480021744425550e-01,
-1.672708830907700757517174997304297054003744303620479394006890e-02,
 2.282910508199163229728429126648223086437547237250290835639880e-01,
 3.985024645777120219790581076522174181104027576954427684456660e-02,
-1.554587507072679559315307870562464374359996091752285157077477e-01,
-2.471682733861358401587992299169922262915151413349313513685587e-02,
 1.022917191744425578861013681016866083888381385233081516583444e-01,
 5.632246857307435506953246988215209861566800664402785938591145e-03,
-6.172289962468045973318658334083283558209278762007041823250642e-02,
 5.874681811811826491300679742081997167209743446956901841959711e-03,
 3.229429953076958175885440860617219117564558605035979601073235e-02,
-8.789324923901561348753650366700695916503030939283830968151332e-03,
-1.381052613715192007819606423860356590496904285724730356602106e-02,
 6.721627302259456835336850521405425560520025237915708362002910e-03,
 4.420542387045790963058229526673514088808999478115581153468068e-03,
-3.581494259609622777556169638358238375765194248623891034940330e-03,
-8.315621728225569192482585199373230956924484221135739973390038e-04,
 1.392559619323136323905254999347967283760544147397530531142397e-03,
-5.349759843997695051759716377213680036185796059087353172073952e-05,
-3.851047486992176060650288501475716463266233035937022303649838e-04,
 1.015328897367029050797488785306056522529979267572003990901472e-04,
 6.774280828377729558011184406727978221295796652200819839464354e-05,
```

```
-3.710586183394712864227221271216408416958225264980612822617745e-05,
-4.376143862183996810373095822528607606900620592585762190542483e-06,
 7.241248287673620102843105877497181565468725757387007139555885e-06,
-1.011994010018886150340475413756849103197395069431085005709201e-06,
-6.847079597000556894163334787575159759109091330092963990364192e-07,
 2.633924226270001084129057791994367121555769686616747162262697e-07,
 2.014322023550512694324757845944026047904414136633776958392681e-10,
-1.814843248299695973210605258227024081458531110762083371310917e-08,
 4.056127055551832766099146230616888024627380574113178257963252e-09,
-2.998836489619319566407767078372705385732460052685621923178375e-10};

const double Daub21[42] = {
 5.488225098526837086776336675992521426750673054588245523834775e-04,
 7.776639052354783754338787398088799862510779059555623704879234e-03,
 4.924777153817727491399853378340056968104483161598320693657954e-02,
 1.813596254403815156260378722764624190931951510708050516519181e-01,
 4.196879449393627730946850609089266339973601543036294871772653e-01,
 6.015060949350038975629880664020955953066542593896126705346122e-01,
 4.445904519276003403643290994523601016151342743089878478478962e-01,
-3.572291961725529045922914178005307189036762547143966578066838e-02,
-3.356640895305295094832978867114363069987575282256098351499731e-01,
-1.123970715684509813515004981340306901641824212464197973490295e-01,
 2.115645276808723923846781645238468659430862736248896128529373e-01,
 1.152332984396871041993434411681730428103160016594558944687967e-01,
-1.399404249325472249247758764839776903226503657502071670245304e-01,
-8.177594298086382887387303634193790542522570670234556157566786e-02,
 9.660039032372422070232189700372539681627783322249829842275517e-02,
 4.572340574922879239251202944731235421034828710753381191345186e-02,
-6.497750489373232063332311106008616685748929419452249544690967e-02,
-1.865385920211851534093244412008141266131208093007217139232170e-02,
 3.972683542785044175197464400756126818299918992482587866999707e-02,
 3.357756390338110842532604766376200760791669954106679933144723e-03,
-2.089205367797907948785235479746212371728219866525211135343707e-02,
 2.403470920805434762380632169785689545910525667396313550679652e-03,
 8.988824381971911875349463398395464114417817949738911101372312e-03,
-2.891334348588901247375268718015882610844675931117463495551958e-03,
-2.958374038932831280750770228215510959830170264176955719827510e-03,
 1.716607040630624138494506282569230126333308533535502799235333e-03,
 6.394185005120302146432543767052865436099994387647359452249347e-04,
-6.906711170821016507268939228893784790518270744313525548714065e-04,
-3.196406277680437193708834220804640347636984901270948088339102e-05,
 1.936646504165080615323696689856004910579777568504218782029027e-04,
-3.635520250086338309442855006186370752206331429871136596927137e-05,
-3.499665984987447953974079490046597240276268044409625722689849e-05,
 1.535482509276049283124233498646050472096482329299719141107128e-05,
 2.790330539814487046106169582691767916283793946025922387556917e-06,
-3.090017164545699197158555936852697325985864588418167982685400e-06,
 3.166095442367030556603889009833954440058545355777781782000278e-07,
 2.992136630464852794401294607536813682771292352506328096125857e-07,
-1.000400879030597332045460600516621971679363965166249211063755e-07,
-2.254014974673330131563184851456825991617915549643308754828159e-09,
 7.058033541231121859020947976903904685464512825731230495144226e-09,
-1.471954197650365265189549600816698778213247061389470277337173e-09,
 1.038805571023706553035373138760372703492942617518816122570050e-10};

const double Daub22[44] = {
 3.862632314910982158524358900615460368877852009576899680767316e-04,
 5.721854631334539120809783403484493333555361591386208129183833e-03,
 3.806993723641108494769873046391825574447727068953448390456335e-02,
 1.483675408901114285014404448710249837385836373168215616427030e-01,
 3.677286834460374788614690818452372827430535649696462720334897e-01,
 5.784327310095244271421181831735444106385099957908657145590104e-01,
 5.079010906221639018391523325390716836568713192498711562711282e-01,
 7.372450118363015165570139016530653113725172412104955350368114e-02,
-3.127265804282961918033226222621788537078452535993545440716988e-01,
-2.005684061048870939324361244042200174132905844868237447130382e-01,
 1.640931881067664818606223226286885712554385317412228836705888e-01,
 1.799731879928913037252154295313083168387840791424988422757762e-01,
-9.711079840911470969274209179691733251456735137994201552926799e-02,
-1.317681376866834107513648518146838345477875022352088357523838e-01,
 6.807631439273221556739202147004580559367442550641388181886023e-02,
 8.455737636682607503362813659356786494357635805197410905877078e-02,
-5.136425429744413245727949984018884707909441768477091944584584e-02,
-4.653081182750671347875833607846979997825771277976548080904423e-02,
 3.697084662069802057615318892988581825637896696876361343354380e-02,
 2.058670762756536044060249710676656807281671451609632981487139e-02,
-2.348000134449318868560142854519364987363882333754753819791381e-02,
-6.213782849364658499069336123807608293122900450508440420104462e-03,
 1.256472521834337406887017835495604463815382993214296088172221e-02,
 3.001373985076435951229129255588255746904937042979316054485183e-04,
-5.455691986156717076595353163071679107868762395367234726592273e-03,
 1.044260739186025323350755659184734060807432172611689413745029e-03,
 1.827010495657279080112597436850157110235336772062961041154607e-03,
```

```
-7.7069098812311962328803727229555197816557699136345657573397396e-04,
-4.2378739983918007995319477680039769781974383025335286618257588e-04,
 3.2860941421367873419837584714059354058233230728296192485236970e-04,
 4.3458999045320033790469926255750760928238096659335755787106960e-05,
-9.4052236348157604218451900983526736478812989800405120915999430e-05,
 1.1374349662125931727361442748666392103398202031356705052872500e-05,
 1.7373756957561893561635650745054059068597466058677720023205090e-05,
-6.1667293164675783721522516684229791521695873072127089817689660e-06,
-1.5651791319951601593074269935782047333781127425799265038320950e-06,
 1.2951820573188775738897112323450681478003957219256825663949360e-06,
-8.7798798733612862768881170461530490539172437604758167892267640e-08,
-1.2833362287517544178196939321140648870750960302647480799767360e-07,
 3.7612287493373623661567116481877433991642393978036290226128620e-08,
 1.6801714049229888855543311836912802459622902476544381148071120e-09,
-2.7296231466329760834493273617391047544432219033177457689388460e-09,
 5.3359388216674899051697832270368045332530111178865863054356150e-10,
-3.6021134843395547037948078109393018472991069702378143341042740e-11};

const double Daub23[46] = {
 2.7190419412828884141926736097033023570983360039209239589247570e-04,
 4.2027488931838335383900343725235114723452155636110034079847010e-03,
 2.9310003657884115147362040189294804278743174606760799595151310e-02,
 1.2051553178397193363060538956118990890042743368917090679580350e-01,
 3.1845081385286523634165277484604721527905750314098304172596400e-01,
 5.4493114787352042826742406724219843875041499248345444954667930e-01,
 5.5101851724191939134527242272125077205141441164787272697178590e-01,
 1.8139262536384001362590983021386149372642607376381755394165400e-01,
-2.6139214803064411188567957352101184139003075775111429873373750e-01,
-2.7140209860784305566040695751847181237636971773810588771134710e-01,
 9.2125407082418052606460309107348942585776480891006300121302610e-02,
 2.2357365824204023171495139608225617176898752527928170948118740e-01,
-3.3037447094289378750066127924630314094616362287312850465516360e-02,
-1.6401132153187592501560578371652760391814511492921129294011860e-01,
 2.0283074575649299748972866075513133234188606107913823103757310e-02,
 1.1229704361810728869507344650756459777546655938697899658745720e-01,
-2.1126212356227241007047832935494670489994438446570584252129820e-02,
-7.0207391574901109462042190119575653438998954999623693532940280e-02,
 2.1765856834499975607768824721687301657994614451567669234975450e-02,
 3.8495332522569199010571543204075960731805646280699208938707680e-02,
-1.8523513650156159797946899607406747828178141761663351959777960e-02,
-1.7537101003035845379158461174086135511479852517265587194151690e-02,
 1.2751943931528286462431574044749471150527505818619977310410180e-02,
 6.0318406500241628162898782060378416408141023142090752337518200e-03,
-7.0753192737061528141940394814665562044932767734838217487400180e-03,
-1.1348654733562516912893371200132867563373937841107869078254000e-03,
 3.1228764498181449974191447651257505224376593936215774925354110e-03,
-2.4650140051635120319404731003753772108625607615761097558411610e-04,
-1.0612312288866513211393576256838056421936486710304250102150750e-03,
 3.1942049270990115036765303596923669909296791700225830076831120e-04,
 2.5676245200787372055638566753766360923148134006641907704354500e-04,
-1.5002185034903409676731632904478322362592778106590686374026680e-04,
-3.3788948341209034342709624526745343309037241089066625103050450e-05,
 4.4260712031092460776218753034409353357018328436546928275398370e-05,
-2.6352078892491862372092259331708978254323352737714584568880970e-06,
-8.3478755678546255443660437488441830867658949744392454092233370e-06,
 2.3975695468402400574037395075256412395095171489808498899864070e-06,
 8.1475748347794477780854430414228814398602882875283560192168140e-07,
-5.3390054052094211545847836828487809650536428593735369457013650e-07,
 1.8530917856339650193536998578646541817287105567025299083041850e-08,
 5.4175491795392787365031761663236855976344961029799770372719450e-08,
-1.3999354954379988451309096873618471032742089934478921203419990e-08,
-9.4728859018120505352215820746734905730920967128220675649030120e-10,
 1.0504464536965434040711051110964385734230689131052559979080400e-09,
-1.9324051113134175421926518996225416123140663896436075077068870e-10,
 1.2502033023510409414332167182175042405414234309951375074047870e-11};

const double Daub24[48] = {
 1.9143580094755136950261383364741155994351720880538467451684620e-04,
 3.0820817149054944362061994245444047209847205561286852705564580e-03,
 2.2482339949716410723584151571848256282267766922319405775815800e-02,
 9.7262235833625196638065457340083559145275044176745785711643000e-02,
 2.7290891606772632687061371344125572687516712634588950986253560e-01,
 5.0437104083992499197718768904028141092468664444418145402820990e-01,
 5.7493922109554199684608079019234070331449459351056229128398380e-01,
 2.8098555323371188334426260851154029418429594759292788832814090e-01,
-1.8727140688515623769818871597757914690602657784416678403079340e-01,
-3.1794307899936273754539484897977075508980877891600251826642990e-01,
 4.7766136843447281879501983230313608663491049940355532007886310e-03,
 2.3923738878031085519732682919458248222148581345123177158156160e-01,
 4.2528729641483832581473644721706452326843432354869515405338930e-02,
-1.7117535137034688968976385150805723939491659423355563979176660e-01,
-3.8777173577920016201775947261995726884464880337507710201902830e-02,
 1.2101630346922423623126373111490622866593770390460068015238260e-01,
```

```
 2.098011370914481534980883827326017063121637262728447783605518e-02,
-8.216165420800166702291466006164189460916816748629968198028898e-02,
-4.578436241819221637997516339765068825260159169893967894877272e-03,
 5.130162003998087915555334881398688958843078494595140394873884e-02,
-4.944709428125628299815920032649550811877887219282751174798211e-03,
-2.821310709490189098113895361900699228886900995412759197674058e-02,
 7.661721881646585897329899904308764405384658404613669817843430e-03,
 1.304997087108573583052494067883717533043101857128653233783396e-02,
-6.291435370018187780721843581169343900864298634085743861509767e-03,
-4.746568786323113800477796959513558401732252800905982385017245e-03,
 3.736046178282523345179052160810332868725126356493155728625572e-03,
 1.153764936839481504858282495202271984454410046682805375157566e-03,
-1.696456818974824394274534636412116243080312601322325642741589e-03,
-4.416184856141520063365958900079406737636243682138363561877750e-05,
 5.861270593183109933716735450272894035425792347806515678695765e-04,
-1.181233237969554740613021227756568966806892308457221016257961e-04,
-1.460079817762616838924301818082729036314539476811023255670666e-04,
 6.559388639305634085303738560455061974369354538271316071502698e-05,
 2.183241460466558363365044032984257709791187640963509380549307e-05,
-2.022888292612697682860859987200455702614855595412267510558659e-05,
 1.341157750809114719319937553186023660581084151828593222893663e-08,
 3.901100338597702610409014129024223853127911530009766793352492e-06,
-8.980253143938407724149926669980791166378388013293887718404796e-07,
-4.032507756879971624098983247358983425236092110387724315244646e-07,
 2.166339653278574639176393978510246335478946697396400359281412e-07,
-5.057645419792500308492508924343248979317507866520688417567606e-10,
-2.255740388176086107368821674947175804005323153443170526520277e-08,
 5.157776789671999638950774266313208715015419699643333784626363e-09,
 4.748375824256231118094453549799175824526559994333227456737433e-10,
-4.024658644584379774251499574468195118601698713554294941756559e-10,
 6.991801157638230974132696433509625934021677793453732225542951e-11,
-4.342782503803710247259037552886749457951053124203814185811297e-12};

const double Daub25[50] = {
 1.348029793470188994578489247159356055370460656508881471268611e-04,
 2.256959591854779520121391049628056149270016860666661928130747e-03,
 1.718674125404015533817186914954848902241194002444696221013131e-02,
 7.803586287213267559750659320481403668422052199257139168386084e-02,
 2.316935078860218199900621518057089104946216881512075361624214e-01,
 4.596834151460945937896973864539659944010260858049947396093277e-01,
 5.816368967460577833534892038757085635755639698734580573323031e-01,
 3.678850748029466984371319740855532278670733841012809062966976e-01,
-9.717464096463814276130048169040892607068486428294030952842447e-02,
-3.364730796417461309562110148848845218930261030262170601615289e-01,
-8.758761458765466140226687673880006154266689569025041229545538e-02,
 2.245378197451017129525176510409543155930843160711989062118482e-01,
 1.181552867199598604563067876819931882639429216001523151773895e-01,
-1.505602137505796309518094206831433270850173484773520730186277e-01,
-9.850861528996022153725952822686729410420350758543226219234795e-02,
 1.066338050184779528831274540522414711301747903916268438037723e-01,
 6.675216449401860666895983072443984697329752470942906490126865e-02,
-7.708411105657419356208567671699032054872853174701595359329826e-02,
-3.717396286112250887598137324046870459877639250821705817221557e-02,
 5.361790939877949960629041419546536897037332284703545849594129e-02,
 1.554260592910229163981295854603203625062268043511894295387375e-02,
-3.404232046065334099320628584033729153497903561399447916116575e-02,
-3.079836794847036661636693963570288706232460663070983852354326e-03,
 1.892280447662762841086581178691039363674755753459524525886478e-02,
-1.989425782202736494289461896386235348901617760816745484282494e-03,
-8.860702618046368399013064252456556969199612331833605310278698e-03,
 2.726936258738495739871469244610042793734119359765762028996059e-03,
 3.322707773973191780118197357194829286271392998979276105842863e-03,
-1.842484290203331280837780430014195744813667655929909114672154e-03,
-8.999774237462950491085382524008429604309720852269895692000702e-04,
 8.772581936748274843488806190175921376284150686011179612908221e-04,
 1.153212440466300456460181455345639872216326644527860903202733e-04,
-3.098800990984697989530544245356271119416614147098459162436317e-04,
 3.543714523276059005284289830559259809540337561365927850248007e-05,
 7.904640003965528255137496303166001735463107762646364003487560e-05,
-2.733048119960041746353244004225286857636045649642652816856524e-05,
-1.277195293199783804144903848434605690990373526086311486716394e-05,
 8.990661393062588905369930197413951232059323587543226269327396e-06,
 5.232827708153076417963912065899772684403904504491727061662335e-07,
-1.779201332653634562565948556039009149458987774189389221295909e-06,
 3.212037518862519094895005816661093988294166712919881121802831e-07,
 1.922806790142371601278104244711267420759978799176017569693322e-07,
-8.656941732278507163388031517930974947984281611717187862530250e-08,
-2.611598556111770864259843089151782206922842627174274274741722e-09,
 9.279224480081372372250073354726511359667401736947170444723772e-09,
-1.880415755062155537197782595740975189878162661203102565611681e-09,
-2.228474910228168899314793352064795957306403503495743572518755e-10,
 1.535901570162657197021927739530721955859277615795931442682785e-10,
-2.527625163465644811048864286169758128142169484216932624854015e-11,
```

```
 1.509692082823910867903367712096001664979004526477422347957324e-12};

const double Daub26[52] = {
 9.493795750710592117802731381148054398461637804818126397577999e-05,
 1.650520233532988247022384885622071050555268137055829216839523e-03,
 1.309755429255850082057770240106799154079932963479202407364818e-02,
 6.227474402514960484193581705107415937690538641013309745983962e-02,
 1.950394387167700994245891508369324694703820522489789125908612e-01,
 4.132929622783563686116108686666547082846741228042232731476147e-01,
 5.736690430342222603195557147853022060758392664086633396520345e-01,
 4.391583117891662321931477565794105633815363384084590559889493e-01,
 1.774076780986685727823533562031556893226571319881417676492595e-03,
-3.263845936917800216385340830055349953447745005769416287177497e-01,
-1.748399612893925042664835683606584215248582345438816346170042e-01,
 1.812918323111226960705459766025430918716233584167982942044424e-01,
 1.827554095896723746537533832033286839689931924709760567945595e-01,
-1.043239002859270439148009137202747658420968144330108510179290e-01,
-1.479771932752544935782314546369458188243947772922980064071205e-01,
 6.982318611329236513756591683950208955110603212379412334701145e-02,
 1.064824052498086303236593797715344405836015002929319291715777e-01,
-5.344856168148319149493577269390074213960237013099439431132086e-02,
-6.865475960403591525454725258715351280947435823354011140858001e-02,
 4.223218579637203541206570902753288247790857760067894456114927e-02,
 3.853571597111186425832144567362328142994885395255438867968781e-02,
-3.137811036306775484244644776337594435094096964336402798072360e-02,
-1.776090356835818354094298625884058170354129044259951019182732e-02,
 2.073492017996382475887790073068984224515077665517103399898854e-02,
 5.829580555318887971939315747596613038479561943085291072787359e-03,
-1.178549790619302893728624468402138072504226527540325463847390e-02,
-5.287383992626814439198630765217969804966319971038003993984480e-04,
 5.601947239423804853206514239940474788977188460452053462770324e-03,
-9.390582504738289646165698675070641765810790863514339205205998e-04,
-2.145530281567620980305401403432221668847980295600748913748902e-03,
 8.383488056543616046381924054554052104937784379435436426690560e-04,
 6.161382204574344193703789012696411561214682388271673214197731e-04,
-4.319557074261807466712901913481943478521991611607433971794602e-04,
-1.060574748283803889966150803551837402553866816191659959347053e-04,
 1.574795238607493590547765666590811258087715699737771458390360e-04,
-5.277795493037868976293566636015627609248847457646525246271036e-06,
-4.109673996391477816326502438997466532822639385119090230965252e-05,
 1.074221540872195031273584409245060623104931330938273936484593e-05,
 7.000078682964986734859102495210684809643657474253921074934684e-06,
-3.887400161856795187587790410706550576033603097954065074023128e-06,
-4.650463220640262639231145944536092973446596027469833860001618e-07,
 7.939210633709952088373459255067360793370284788682979065122810e-07,
-1.079004237578671411922961583845716126060658213943840375162654e-07,
-8.904466370168590769052983362721567202750591914741016835071257e-08,
 3.407795621290730008673832107214820587991557116806912418558069e-08,
 2.169328259850323106986222296525930099935873861026310788086221e-09,
-3.776010478532324328184043667556576385639846460337894963138621e-09,
 6.780047245828636668305808192607091517605349478677442468580825e-10,
 1.002303191046526913509281844136258004034177309673269533418644e-10,
-5.840408185341171468465492447799819262905317576847426970757700e-11,
 9.130510016371796243923232926650252570239054815939483900056681e-12,
-5.251871224244435037810503452564279828539007071678724285717464e-13};

const double Daub27[54] = {
 6.687131385431931734918880680779563307675740731544063787599480e-05,
 1.205531231673213234251999812212394463872002561229330125152073e-03,
 9.952588780876619771874091297340545740163119816300838847749336e-03,
 4.945259998290488004302995584228917712171023349013386944893643e-02,
 1.629220275023933206396286389387812803673796872000118325233533e-01,
 3.671102141253898226423388094379126394383458407087000700420400e-01,
 5.538498609904800487605460395549044755068663194750017660900436e-01,
 4.934061226779989979265447084358038959373468583404767251300717e-01,
 1.028408550618229112710739475157388764479351682549490307668477e-01,
-2.897168033145948463175311101489473923261698802610323264603418e-01,
-2.482645819032605667810198368127693701263349361209208170092197e-01,
 1.148230195177853576326445213787661879970642975306605349249036e-01,
 2.272732884141708265275037216925482827043581894357907763081103e-01,
-3.878641863180231062443346843661817078060143110529946543683356e-02,
-1.780317409590085821070366277249759321269342801053489323888575e-01,
 1.579939746024048431173907799261019471878724997312653292884660e-02,
 1.311979717171553289711406975836688896451835867594492827800969e-01,
-1.406275155580876537026622167053147161846397735962817855782362e-02,
-9.102290652956591798241345515773322449830692586525337562864481e-02,
 1.731101826549371089085675445961947677452358872325373949295769e-02,
 5.796940573471798814748840657698008349462526768238833307489106e-02,
-1.851249356199807710545837861298826718763077900221574749342712e-02,
-3.273906663102087145481936428049519742538150452785563039743756e-02,
 1.614696692239566682272152627542980896527822528487665111124260e-02,
 1.566559564892457873003263983940819950829497022298967052103291e-02,
-1.157718645897628140054089958116866381056430680879332334217267e-02,
```

```
-5.862096345462925972966025215266179082657169806555503857975278e-03,
 6.856635609684880675273184141746359000591385833807880272568038e-03,
 1.342626877303679609082208800217479591902967766971379107017011e-03,
-3.332854469520006162763300141047111065412307706449049389557931e-03,
 1.457529625931728587128588244152604734177322144376309490881599e-04,
 1.301177450244135139135787970279897042994109161268159963884641e-03,
-3.418351226915427611946547437228006377896519777431057005796358e-04,
-3.879018574101327604369144470124819695479087900682219330965466e-04,
 2.019719879690326857104208791272390315160018069955787875123234e-04,
 7.660058387068576876674274961751262847965101108848090019821555e-05,
-7.711145517797584208411720507329584053382646435270054267102827e-05,
-3.517483614907445391752737841583832374184046409747387149129674e-06,
 2.063442647736885318487206413360228908558806028468062177953960e-05,
-3.901164070638425528170558032557368703418425915665413541985623e-06,
-3.657500908187104997045760131046655906827644494899206692043298e-06,
 1.634369624725637835424610743915128591988676092276368687669255e-06,
 3.050880686251999094242671997731089918322345713516567387655763e-07,
-3.472468147394389269364673179891460601330730511237974736379548e-07,
 3.286558968055159530983261866450459360074591641809187825408848e-08,
 4.026255052866908637178682747490340533992340623231336911661711e-08,
-1.321332273990056558848617809101876846857728483295631388083263e-08,
-1.309465606856955151282041809232358209226373823424148862843577e-09,
 1.521614984778521740775073159445241799352681846880808663329946e-09,
-2.415526928011130660506395791946234018673860470542996426005750e-10,
-4.374986224293654395069947682013996351823060759948583134078918e-11,
 2.213662088067662485181472969374945928903854605356443772873438e-11,
-3.295790122476585807069953975043096139541415768606924980926275e-12,
 1.828188352882424933624530026056448539377272017834175009418822e-13};

const double Daub28[56] = {
 4.710807775014051101066545468288837625869263629358873937759173e-05,
 8.794985159843870273564636742144073059158975665525081816488582e-04,
 7.542650377646859177160195786201116927568410621050693986450538e-03,
 3.909260811540534426092083794403768111329778710541126982205076e-02,
 1.351379142536410450770749411679708279921694061092200363031937e-01,
 3.225633612855224257318486139030596702170126503618082416187649e-01,
 5.249982316303355562348293243640252929543774162151269406404636e-01,
 5.305162934414858075256978195354516449402692654391295761050628e-01,
 2.001761440459844380384404537971725815970574972480152145882083e-01,
-2.304989540475825257279397658067038304888129374484095837624889e-01,
-3.013278095326417816909366061441334075444383937588485826752087e-01,
 3.285787916338710468450547883547348694255260871071954509422161e-02,
 2.458081513737595535752949960866466132239832334168533456626848e-01,
 3.690688531571127205290633425993077868843846977265847006108551e-02,
-1.828773307329849166920408764650763092868965221608724574218473e-01,
-4.683823374455167616514752420549419665215987106243491879971921e-02,
 1.346275679102260877490923315484152662987698625205479167761416e-01,
 3.447863127509970524678534595639646616244376966117385829345554e-02,
-9.768535580565244174963692133038973587005628990493154911133358e-02,
-1.734192283130589908795581592406238282930530566316914040035812e-02,
 6.774789550190933956165341752699717255041141690153626336867769e-02,
 3.448018955540951137600471926079622335842207388713342609755316e-03,
-4.333336861608628393863254980828284403766309203453808666888800e-02,
 4.431732910062988320487418656322338284504389482966303454010563e-03,
 2.468806001015186586264188361362046240243934625858343309818244e-02,
-6.815549764552309639259447104811254179605050667281644254737890e-03,
-1.206359196821849005842466619530619474644989878503490321948471e-02,
 5.838816627748944864497370576838809711476027837762897602935327e-03,
 4.784863112454241718009916669120329848973107781600157214960003e-03,
-3.725461247074254799171427871442937099025589672466088044410521e-03,
-1.360373845639692436577650137133777929659265166644839235882291e-03,
 1.875998668202795626152766912508562385106168761893900192731562e-03,
 1.415672393140464257573780581396205840941849282748250523509874e-04,
-7.486749559114629991320679819683227355746847370960399216568306e-04,
 1.154656063658921251969297916771881248142872975490882572741198e-04,
 2.295790982233456202366621544054366855729175050420515776344878e-04,
-8.903901490044488099517361247378396756893227855233897357882978e-05,
-4.907713416190250858324783990436748073854807494400738311968278e-05,
 3.641401211050802781223450761733180188911730291497201507086247e-05,
 4.638664981394294654002871426476885751050837817671843706915388e-06,
-1.004326041333422601781848560432120920634648692782357855473103e-05,
 1.247900317574834146052381692752796047052443265982232422642017e-06,
 1.840363734517769191684379309039277810350620305330900536404818e-06,
-6.670215479954892588747450458085225880096882699397256774967304e-07,
-1.757461173209842779903676264971918635870906983281392939812547e-07,
 1.490660013535362170989340065033061951960933954388633507264360e-07,
-8.262387315626556965966429243600984899650039704831080988658278e-09,
-1.784138690875710077191713941441263246560738410213624546116655e-08,
 5.044047056383436444631252840057862002264087720676808580373667e-09,
 6.944540328946226952976704718677697525410051405055662575530111e-10,
-6.077041247229010224760245305596307803830053533836849384680534e-10,
 8.492220011056382105461206077240377024404404638947591299761197e-11,
 1.867367263783390418963879146175452376940453585791428841004699e-11,
```

```
-8.365490471258800799349289794397908900767054085216008197372193e-12,
 1.188850533405901520842321749021089497203940688882364518455403e-12,
-6.367772354714857335632692092267254266368934590973693820942617e-14};

const double Daub29[58] = {
 3.318966279841524761813546359818075441349169975922439988843475e-05,
 6.409516803044434540833706729120596322083061716935004987374676e-04,
 5.702126517773375434760843998623507494914551464968126455168657e-03,
 3.077358022140837676716707336516751814713312018344719150923618e-02,
 1.113701169517405304762186166370327770191325772342190715118617e-01,
 2.806534559709829376968881262770480606500920092398534229615289e-01,
 4.897588047621993143592705932993573539235839610055331620240518e-01,
 5.513744327583751951223746071670135992466984391233429663886536e-01,
 2.891052383358291634605691113586264061513180158354460952469246e-01,
-1.540287344599000542466293779503370141731339982919280951230240e-01,
-3.300409489175880520295083779487012611959310539629627124613719e-01,
-5.570680007294085781514541931715795784309410235726214400350351e-02,
 2.361052361530259415983110734054626770649468357328362426830433e-01,
 1.124191748731883764769740670535880543076817816861518667898467e-01,
-1.608779885941877360771615465531852333085159940159968393590303e-01,
-1.078459499387214201077881957354707913786241153934264316589273e-01,
 1.144722958938182579734135930060053286267822797640393386903440e-01,
 8.322074716244975790297348835032537357891920536002627784941129e-02,
-8.512549261563550232832311331420804581881235448862834507281486e-02,
-5.502748952532572320924541450626650067707344725344841099873446e-02,
 6.347916458421186633577789314698972361081611994794140119302163e-02,
 3.053154327270413646637328212093941030592133225231728964047047e-02,
-4.518798127778834515979704475304405691390090327474972089790857e-02,
-1.291714255426679462966473962555410660387671182428076570686472e-02,
 2.947043187174764111028122319949903667638786379520519899154373e-02,
 2.648327307678167915542397563479749119673768286990136051577167e-03,
-1.704122457360668969234196743407615179099529206118693044741086e-02,
 1.737880332720511164430027824345354801611373419264590068097416e-03,
 8.469725493560752287772961661104710791306496373354237126998903e-03,
-2.550807127789472659145072247724735637183590942511858255354005e-03,
-3.473798989681100630649790255076233970957721666820195620598374e-03,
 1.877120925723650133179338154344873477230567340668548016358682e-03,
 1.087053942226062966738944397844498417945523630053411148182206e-03,
-1.000778327085680541055696707760062870925897014530348262794137e-03,
-2.000711363076779808296301110796026470163110202848894744316755e-04,
 4.111283454742767033424740543004041500054889660665367490129376e-04,
-2.292018041214499897382298271438084577065170236103859181134525e-05,
-1.293044840080720609161466939678226852440475312744714379499074e-04,
 3.645026068562774967665464216602750761690984830805534178557146e-05,
 2.913344750169041218495787251929571015775436967652945386217480e-05,
-1.657328395306616289863396387854880512976861409870690029695161e-05,
-3.593644804025187638066915189731950450034629392522542962477168e-06,
 4.750609246452552850197117564759363194953518317428400241629683e-06,
-3.029054592052818286474228294307141792053791695855058563299597e-07,
-8.975701750636280734511651941681818767895052287332471537510510e-07,
 2.633898386997696553900967704111473475368019612368922599394214e-07,
 9.387197411095863026484410601284876812292554863800653292318725e-08,
-6.286156922010786166768503252870590953166867739448102804392389e-08,
 1.076591906619196137385201975028785139607670319821266803566785e-09,
 7.768978854770062238895964639391324551611701293594055935346266e-09,
-1.893995386171984147774611076618946011337498790609031626697228e-09,
-3.426800863263089001811012278889864200550342566386405676893537e-10,
 2.407099453509342962399811991929330725186626582891090462239366e-10,
-2.940589250764532582888473974638273664244682541297835986306504e-11,
-7.832509733627817032356556582819494794884131433810848844709881e-12,
 3.152762413370310423797539876893861621418382024668704492620948e-12,
-4.285654870068344101898185073376307686875386259541180967347399e-13,
 2.219191311588302960934661700068023727737812918006011019184982e-14};

const double Daub30[60] = {
 2.338616172731421471474407279894891960011661146356580425400538e-05,
 4.666379504285509336662000111055365140848987563882199035322085e-04,
 4.300797165048069510045016757402827408493482974782286966500398e-03,
 2.413083267158837895194919987958311943976725005113561262334092e-02,
 9.123830406701570679321575555085899708564500191080751595642650e-02,
 2.420206709402140994467599658342919512318194032687898436229538e-01,
 4.504878218533178366981351802898336415314944375740699506554771e-01,
 5.575722329128364304078082520999850413492571645754785374629734e-01,
 3.662426833716279793144871151369089533016299234992584741629624e-01,
-6.618367077593731501909741041813726474911212544474895441395148e-02,
-3.329669750208556069196849320598850505877494561268613506392514e-01,
-1.419685133300829310219026267403758254954270602825020111483505e-01,
 1.994621215806643032428990062111230223523226088131364328774921e-01,
 1.778298732448367361280250921330425046260289700971176750362566e-01,
-1.145582194327077814891518778613672243404957549114393749173137e-01,
-1.572368179599938126878197378886501553251711910617673398124611e-01,
 7.277865897036442699893544326605244235248713804556715604416632e-02,
 1.227477460450093778691578797698150091624353365248212907325446e-01,
```

```
-5.380646545825707676022015051837304300338645984615639237930800e-02,
-8.765869003638366048026572053699028353846982304851342479893827e-02,
 4.380166467141773250305407710250135373016604593736480428415303e-02,
 5.671236574473569492590636983030617493807140224924978946302257e-02,
-3.567339749675960965780819743176056734137251336781389369397564e-02,
-3.226375891935220815954913483392725682165778426411705216010280e-02,
 2.707861959529418272206848318420006522973840949600186710327776e-02,
 1.528796076985739546052896626042375110302102640936712142026221e-02,
-1.839974386811734118728169880549148389603890445324127330811811e-02,
-5.296859666131086629169938675330494864053932988161015674773617e-03,
 1.091563165830488927536881480211929049886878831313700460017968e-02,
 6.196717564977244383592534999284255315694546230739551683085460e-04,
-5.530730148192003288871383856487027893918513053091795443517653e-03,
 8.433845866620933982126003584365932145598126087481400294999080e-04,
 2.324520094060099304385756339638431339131122661576649123053845e-03,
-8.609276968110423879660725173525347077801305237644122054954659e-04,
-7.678782504380918697963922441514742758516706160788123977340073e-04,
 5.050948239033467796256544554086554367969638627715114003635557e-04,
 1.724825842351709725545759714374272164367933578194910678479473e-04,
-2.161718301169633804271038862087964094429005266172702380483361e-04,
-8.548305467584070994787824796256108217987765582429940610377190e-06,
 6.982008370808327851082027193100914402221658444151889697045071e-05,
-1.339716863293971629296314599448901465078920406443516550195793e-05,
-1.636152478725426488654528710478856195004608401773950511915162e-05,
 7.252145535890469015723401169934327900622894130695550273452916e-06,
 2.327549098493686509557358103785598216688723737824121617676858e-06,
-2.187267676996166416699555236143059249832615777542412142603694e-06,
 1.099474338526203304286307383463498542376432972308342428764576e-08,
 4.261662326011572446469849114416378817419458434583398455985144e-07,
-1.000414682354500898864979332965559934104686157639553850670490e-07,
-4.764379965139453357729154748688006975561934425368712852985388e-08,
 2.605442754977625431940885841950955928085338672381046225838880e-08,
 5.553397861397053982967618072672572206490972606026556946910028e-10,
-3.331105680467578245901976412732595596538702049437802824373020e-09,
 6.984862691832182584221096665570313611280449991512869846064780e-10,
 1.613622978270904360610418704685783656905979134344922647926295e-10,
-9.461387997276802120884525814092001871993910062127702293573920e-11,
 1.000105131393171192746337860330428369495110180346654025287492e-11,
 3.239428638532286114355931428908079297696045600279108835760520e-12,
-1.185237592101582328254231496310584611948560976394420324137742e-12,
 1.543997570847620046003616417646988780670333040868954794039905e-13,
-7.737942630954405708679963277418806436871098329050829841696327e-15};

const double Daub31[62] = {
 1.648013386456140748122177817418358316441195236228590958603489e-05,
 3.394122037769956699157160165352942212213928231154233571163033e-04,
 3.236884068627721221829662672296912258338131668810067169630813e-03,
 1.885369161298591269159568944275763468999829139547989648553486e-02,
 7.433609301164788697908776495388047669378919816041031344650271e-02,
 2.070128744852353286198055444111916450619762837756134323019573e-01,
 4.091922000374278563928213235836188963704298775635493549519369e-01,
 5.511398409142754983590484577074663132074992263886810324421617e-01,
 4.294688082061372955430413148799008354573408538414331312236645e-01,
 2.716921249736946422305354732634261873401679092095992827198308e-02,
-3.109551183195075186926560285811004715398678229333522634202008e-01,
-2.179784855235633521693544507220105631639547435903112747133934e-01,
 1.401782887652732681656253206993073895422881511380152633441096e-01,
 2.249667114737370933697297905066886078307490136415302624018330e-01,
-4.992634916046823977000579399730138693074543903234092797936484e-02,
-1.869623608957154494374577196258383009208655076187653847079167e-01,
 1.543698842948893409652995335281236231845293548571166883219023e-02,
 1.450895009319931981518942907854879059128872873116921504156674e-01,
-8.139832273469236863527708715566588550006680549152344840146851e-03,
-1.076127733234956326668605511648013952380301953590447106075614e-01,
 1.094129745236496925725237900637802669504835743555466811796369e-02,
 7.535361174328140695528289751109133941376701984419452638686226e-02,
-1.488002661810482202699555987503429289100801979910046913257306e-02,
-4.861907546485433003537603385831190109391263542044516048871113e-02,
 1.615417156598591113619453864586701665635869166193865651960591e-02,
 2.804761936675616906861927211659154977049392281479113764697785e-02,
-1.427627527776351943309800140756746087215016194775579070599004e-02,
-1.390055293926652880755898888934447671732373519028670201124816e-02,
 1.051763948737184089128633441244991643331033825102031908858652e-02,
 5.516163573310992566561289762241160214476622662764637181816550e-03,
-6.520852375874612553325469682628530079210293774541131381751695e-03,
-1.428264223218909891400516038687842292177211292295049238921068e-03,
 3.393066776715931928419358796960612411097347419792355896915546e-03,
-6.397901106014600492881202314307290077992972755016494062875201e-05,
-1.459041741985160943114515221598080223845239255190055621901681e-03,
 3.431398296904734438118401084929505912208229684629857530009147e-04,
 4.998816175637222614896912406679513231966722440032799024979502e-04,
-2.396583469402949615285646688069476140260781708006174912535660e-04,
-1.243411617250228669409179807383399199879641177993453588807726e-04,
```

```
 1.089584350416766882738651833752634206358441308880869184416670e-04,
 1.501335727444532997071651937630983442758297688087711521441229e-05,
-3.631255157860086164261313773172162991107348698083164489165837e-05,
 4.034520235184278839752741499546098778993926344831736074409765e-06,
 8.795301342692987765440618030678349427367022581211855857458220e-06,
-3.035142365891509630069007852947057220760887215249503512783023e-06,
-1.369060230942940782050489751987123955074404782177163471279285e-06,
 9.810015422044371573950976088058064384946146188110905321673802e-07,
 5.327250656974915426977440959783080593776012130063170688309127e-08,
-1.975925129170206248152121156696590501303803187231928513867046e-07,
 3.616826517331004805247567218405798591329788122337274956172315e-08,
 2.328309713821409644308538885893299211419485396781066807770 82e-08,
-1.061529602150252306500404266150823962402673780484965538270541e-08,
-6.474311687959861398702581539341954438747926255671605657095807e-10,
 1.408568151025177427076547804944585301332087108125727813194374e-09,
-2.524043954153353306183643702933218308617979467184848456565837e-10,
-7.348930032486263904766913919653624379586487437915175106407348e-11,
 3.692108808871129411604189196259677640440919369478263728899602e-11,
-3.327008967125979929910636246337150851642079794871116041187279e-12,
-1.324334917243963163878274345609465717294426628053460151843705e-12,
 4.445467096291932163298411852093011459626037560439178917611592e-13,
-5.559442050579014337641375730083534521513818164827556763756543e-14,
 2.699382879762665647295493928801387173921314576598505507855504e-15};

const double Daub32[64] = {
 1.161463302135014885567464100760659332951431420121048996305591e-05,
 2.466566906380903352739104211274667134470169443886449124673996e-04,
 2.431261919572266100780423071905958127811969678055971488060574e-03,
 1.468104638141913563547809006402194831107662001343421893488086e-02,
 6.025749912033537081745451975527967031851677384078997261920024e-02,
 1.757507836394388988189299915753348505208376399651864661397588e-01,
 3.675096285973496361995340339143234125206079560406868595968025e-01,
 5.343179193409538322901117858552186425529774700290587495921679e-01,
 4.778091637339484033555130814414794130354053753675509287934741e-01,
 1.206305382656178269538098710665261299391507308342013788891222e-01,
-2.666981814766755535489784087869865024226542605534080371507405e-01,
-2.774215815584272153338153320303401666681294506143291967655666e-01,
 6.471335480551623831000090095167664918448659157720155321560811e-02,
 2.483106423568801736064852157222867588791898170114101300999760e-01,
 2.466244483969740441701479334808723214802614938081258920635302e-02,
-1.921023447085468984341365278247990525863123891147783426068990e-01,
-4.899511718467173853355943225576377418394280156945986899417475e-02,
 1.452320794752866460838830744051944832326998342053148426312341e-01,
 4.440490819993974022640619534046603571086531544468421519143629e-02,
-1.094561131160893831027722774343269232755171130623890041619420e-01,
-2.962787250844770491204452379051215505049068645551070779367843e-02,
 8.087414063848395744090831590426327690818854671836423275412813e-02,
 1.410615151610660772869738802931740150275269382463799031013905e-02,
-5.692631406247843550478416271158537960555270097953330567652364e-02,
-2.380264464932573834443178362086503847328134994591954135879789e-03,
 3.705145792354468010437633458013030898015496905609424004450953e-02,
-4.145907660827218781460700428862611061267328108653649653634276e-03,
-2.166282283639119347634778516947485598599029367518033869601702e-02,
 6.167527310685675112579059689520105004744367282412921739811164e-03,
 1.101740071540688116532806119564345712473051769079712407908648e-02,
-5.411568257275791208581502410752383050600045942275647685361370e-03,
-4.649216751184411528658094984504900172989190128905887602541396e-03,
 3.627224640687864960122122984391704782343548385375321260251988e-03,
 1.468955100468467772528811782840480639166582822577191079260543e-03,
-1.964740555821778254183647540656746450092725858126595984907304e-03,
-2.211678729579097916278097586914956834196749138610403102772710e-04,
 8.673058518450555343925662389563539890596549655683386287799624e-04,
-1.024537310607396186949656796812972062290796122915930356634122e-04,
-3.059654423826911750479261161552574500739091332121504634422577e-04,
 1.053915461739828114700905192091104141076083602686374410146603e-04,
 8.103678329134838389828091896334156224227821362491626044950428e-05,
-5.259809282684322782648914338377962890245975842272425408122506e-05,
-1.294045779405512723950480259110995722517019870286295908085366e-05,
 1.824268401980691220603850117995712615809177092802967489081228e-05,
-6.361781532260254953363913076575914206506177493714496098327288e-07,
-4.558309576264423135123964145585288808181431652781253437738445e-06,
 1.202889036321620990296134494079846952404216422923750605507047e-06,
 7.560047625595947819392627283726711361273296630256477108501994e-07,
-4.285970693151457255418342315045357407199066350632593899896712e-07,
-5.003361868748230293692887222336390314786090450819216035110269e-08,
 8.965966311957728376981484572655177545054433542721057470726361e-08,
-1.219924359483373093110396748985081720383992859961285213840740e-08,
-1.104383021722648979552131128575075255513372249283096583736746e-08,
 4.250422311980592983740943309197245384991941251563471671065543e-09,
 4.384387799940474369553236949848427579687147486892033587998023e-10,
-5.881091462634605628881794361152305108432139465417759716875076e-10,
 8.904723796221605490455387579189371137903330749397374037644960e-11,
 3.263270741332907875981844980104948375955551273115386408552080e-11,
```

```
-1.430918765169202320188022211739750594608742928641485026836608e-11,
 1.075610653501062115165734990153347111902874668945095034791947e-12,
 5.361482229611801638107331379599434078296259332654994508124989e-13,
-1.663800489433402369889818192962259823988673359967722467427927e-13,
 2.000715303810524954375796020597627467104635766752154321244151e-14,
-9.421019139535078421314655362291088223782497046057523323473331e-16};

const double Daub33[66] = {
 8.186358314175091939858945975190102731733968885547217619434602e-06,
 1.791016153702791479424389068736094134247294413108336017758506e-04,
 1.822709435164084208084617771787691709255513374281497713580568e-03,
 1.139594337458160925830840619716397130445853638884729488329 32e-02,
 4.861466653171619508385707681587366397164931431125053574327899e-02,
 1.481863131800528081784673514426737436792606299953305691300616e-01,
 3.267181301177075783930752787756046348844272437670999719562429e-01,
 5.093761725149396552227892926384090200953139820961482931291482e-01,
 5.112547705832674655425831875568453973369927971748064975152374e-01,
 2.095823507130554216526494469993023406452629154801126958766008e-01,
-2.042026223985421049629055102642279430174095014493415546881477e-01,
-3.159974107665602561905181464284910961862968513875028980451424e-01,
-1.927833943695275915600583425408664108893845271616240406358226e-02,
 2.454206121192791114179964351253140999836791489738418857473689e-01,
 9.985155868033815698139640215477639365289384281516885362929979e-02,
-1.714280990518593279308738113273443832545615219650436927029674e-01,
-1.108441331167107910806084983056783194189909198734302929909672e-01,
 1.219678564037346149389134584371009777591763921148126952722200e-01,
 9.478808805061595889263191779090571160237408179346345390888721e-02,
-9.114696835133148913093153757138373418923462847746880902676089e-02,
-7.030248505405615921453280814171665167171986608963193275084895e-02,
 7.019114394099653254998935842432841393915841096633514680190145e-02,
 4.573456189389667743139040427641638967843459421665709740086516e-02,
-5.347125133582228919431110824663168583260050383336359554980188e-02,
-2.524858297747649929258392207837724793937727346177294684700378e-02,
 3.868706076024496481748675031852528047303323816250150793091832e-02,
 1.070326582001954942654534968137727769698168853186071888736311e-02,
-2.572876175473297336123211392278301875687760837710204579628265e-02,
-2.167758617353607324783298657172830203896433848418061622436727e-03,
 1.531695411585766548347442266431874060229304787191589430967538e-02,
-1.594288782414604768637856446111392724059836934455189837500244e-03,
-7.953540387057939240459305406538116220678495240302592677582773e-03,
 2.389062408165908575935815973439728988151836094753689966108405e-03,
 3.480800953405711999411461002429227385937942254778524257436278e-03,
-1.860718214455795912074482150710567824317228203897000129729967e-03,
-1.204309257604658876916644980097327372892008586047095719636829e-03,
 1.074380696351291355073899234941719080473877020595209197706651e-03,
 2.727305847336937211749282358350196461733595290569540045817329e-04,
-4.908329007590351474487792254066540683724948757382104652497458e-04,
 4.393166251766185755059005296958129844094063524324718175254673e-06,
 1.780431898251245351831728023200069586928513661382622116969992e-04,
-4.160438516273709306234368807933932360567787692918883118883736e-05,
-4.929564423417301834310231482621574127409950921583062559483686e-05,
 2.423335398816890365621188379922041046073808819182024026589770e-05,
 9.070805757828453800203677464921508178468256685438211818575040e-06,
-8.866121366757736169176034432364298134186929098274651022820760e-06,
-3.607516102879771631230351118595069330196155459105589342866625e-07,
 2.288371276141527305481395545993763010565968667577768164201792e-06,
-4.426923407952870147984002129341809185622768353983550670755106e-07,
-3.985791291985944076942626511739220753169387460984290019185514e-07,
 1.822443332571053437467128998002798233969112236553215291639303e-07,
 3.377972703730854377516206663481869099376154259897212784144779e-08,
-3.987838198518880722819502850814936369197384392561970319349663e-08,
 3.672863576838181340505563759379169099717712645283448779390320e-09,
 5.111211857347453839549366593998758891130921028374576213256027e-09,
-1.671392677251932495173219614104411841891545601521784559793012e-09,
-2.496402105246193648073519269370197331176405371538404298745013e-10,
 2.426833102305682309891302883361232297664099485514601790344279e-10,
-3.049574453945863430361296931455141500128170151643206937547928e-11,
-1.420236859889936792437077844940412749343225644487770840543290e-11,
 5.509414720765524548752673631197714447818740985929081064907524e-12,
-3.343481218953278765982532722689984725170758193566174566492199e-13,
-2.152488386833302618520603545685994753329478275805993737095214e-13,
 6.214740247174398315576214699577230693021307854673557214652751e-14,
-7.196510545363322414033654470779070592316600780697558361083151e-15,
 3.289373678416306368625564108782095644036415401902518812978798e-16};

const double Daub34[68] = {
 5.770510632730285627466067796809329117324708919047900817738025e-06,
 1.299476200679530037833484815390569400369432658207722720405084e-04,
 1.364061390059049998200014449396877439591680435610837369411339e-03,
 8.819889403884978803182764563095879335330977939541630862804757e-03,
 3.904884135178594138905026219591569204043816577941517019631916e-02,
 1.241524821113768081954449898210969172708199672428635378051285e-01,
 2.877650592337145629334256618087718872558560120999651277991839e-01,
```

```
 4.784787462793710621468610706120519466268010329031345843336104e-01,
 5.305550996564631773133260223990794445605699030503652382795600e-01,
 2.903663295072749510455945186199530115755664977934564128822650e-01,
-1.282468421744371672912377470485584276127749329437486286508 24e-01,
-3.315253015083869417715548463087537345035828886426345397256876e-01,
-1.038919155156404718287260506925867970596448618647006698388596e-01,
 2.169072201874275950610018667099322465619408030256534197819784e-01,
 1.666017504122074437311574334509261366682993700573488534577890e-01,
-1.273373582238011562843862636988693890108793629966541695807247e-01,
-1.609249271778668063014799490429649196614628857267382976958607e-01,
 7.799184693794810738265349531832015087096882277333968473726399e-02,
 1.341259602711361284802399913977387999358280900708582462625539e-01,
-5.448296806413904636632671383140642554265865948686157271017286e-02,
-1.029475969928140852342073823689090498245496056845473569066667e-01,
 4.357609464963129726428486610925800727137724136370669421246609e-02,
 7.318523543679560555546221335452045680757998947493883124934567e-02,
-3.701283841786244960356402125554190040750079009127461655784927e-02,
-4.743855964527776247220681410983851377889756018716427358008296e-02,
 3.073974657395934459931226513844134346305562928466993208164603e-02,
 2.722835075635419610095839895805858855202745897718117731496534e-02,
-2.367173792282636485046786438094940427456079528043555566867110e-02,
-1.314398001665716086105827506126287041342680578404007359439612e-02,
 1.640937419986519252112261495537409592363156309874473310057471e-02,
 4.713649260999809905918876125437488856235874027077755004539205e-03,
-1.004550670836151917439146861146431000364858401181337134891421e-02,
-6.194748845153872839014356621835501857322345445234809347431098e-04,
 5.334950768759936032170270195983921511565539100791906952901398e-03,
-7.692127975067836975989490900561029844887285335804349474993607e-04,
-2.399453943537055863933124827688081952701780599883067560501870e-03,
 8.589959874363661955444898475746536583497522107459291718900058e-04,
 8.751999064078688732610570055224339733760304773327228476255647e-04,
-5.527355762144197975516415296735124460550632283763688359649888e-04,
-2.326732140233531635428863212833942245597361085708567528230733e-04,
 2.650772397558057819755811309071002543822145660933016957735937e-04,
 2.660050018453441903046828468025589086403126180798464347801678e-05,
-9.914697770780134603580350758869378471802751837608461971022567e-05,
 1.353117227249649581251887376414486225127346352042209141315562e-05,
 2.844951419697807376503080001943765930601242225183893658540032e-05,
-1.057657494257950623848316304755218120233253479317574337409622e-05,
-5.710826510998303938275050074333400305512451419983646591762318e-06,
 4.169871758547028398316761659984928804362023643629741358799744e-06,
 4.979718101421307748081857636471761057429219265531618602960147e-07,
-1.116306534817008428597995070751765080383261658112656948526954e-06,
 1.448195708333185127061180618150009526758658641231104901703561e-07,
 2.025990666667859216690536885693725545344933235432307649205497e-07,
-7.526701740412589411177481797841044281662555785969415398369019e-08,
-1.990346501531736915866180448337614967570744211158241514589121e-08,
 1.740423332936068076497051274445147160190783847854409836489662e-08,
-8.665744261368722215864741166245385888818567571145958531936939e-10,
-2.316501946995482751582294240136010067415084499025753117941001e-09,
 6.446378210323402313101214894500231181606520211579581132442548e-10,
 1.300410318609415248880403259300467720631189120978928377152233e-10,
-9.904774537632409015479530333979124540183199174591377762845227e-11,
 1.004208735461769864836516428998306778031143650101842361622330e-11,
 6.080125354000167254059025929915591291115751734288584563131636e-12,
-2.107879108915301546285370395443778864676275235126044599683271e-12,
 9.799451158211597727901178520526388692140586041163624252991805e-14,
 8.579194051799733179793112298652600511486581216528683482143106e-14,
-2.317083703906408481078257081903089523234020423092175261925515e-14,
 2.587338381935699555813538163144986688834142571207152879144731e-15,
-1.148944754480590128244815794312606245147888158018823490936280e-16};

const double Daub35[70] = {
 4.067934061148559026665247110206084571051201477121972612218005e-06,
 9.421469475576740631603027533116630224451049736050903361458759e-05,
 1.019122680375098109319314672751485080202557607467199213778085e-03,
 6.807292884319132011971333979015625113494050642797397817625326e-03,
 3.123628851149071453063391210769353068187088999495893257051179e-02,
 1.034044558614783789938787754929279183985553322796063517049140e-01,
 2.513073789944933128513251971488905042866779761014740192816902e-01,
 4.435927392240354378183910489448494594782039032807956294826105e-01,
 5.370084275091661028670690231716974547580034932361053607723887e-01,
 3.603456405180473278744458573988718422538114217890792270621563e-01,
-4.388388187393404111343479394097224312100349011932028865098625e-02,
-3.238228649121161212147302807993176715625480327235512530593160e-01,
-1.817869767667278325788350264528191676841493369460849123538616e-01,
 1.660413574907809195438433327470947940538097914525298064477785e-01,
 2.172992893210892977675493456199559114036326358517672106972956e-01,
-6.526287131067753892154895911331108284007380738865652420304233e-02,
-1.919195892985939528760786800798636198516495957924798820500876e-01,
 1.930954466601835091947734585938109944647435243484967057775110e-02,
 1.552924803962371144206753760712566993987319378965231186477630e-01,
-4.752680834111350445288110998030979143710864689041902167119118e-03,
```

```
-1.2058552264339355450765894807049577226353244568123222150437989e-01,
 4.7342291726419487632939803149922132939717706954806167898283840e-03,
 8.9913547570729544178653741952619629836440489982182339004818560e-02,
-9.3185589499039248378750028236175042272465621526718945795043780e-03,
-6.3356037440443466120988875340205457057316717180579648020066710e-02,
 1.3228549585036555244559298476051107196487468904973568082893020e-02,
 4.1254693064705092127497508142991266561515048058454179946514170e-02,
-1.4366839784220071821040251732140127977889048942917163734935250e-02,
-2.4169497801660267402948806817310840912645331688167462275370300e-02,
 1.2766456715656744194039180187424327149736565982279398249400350e-02,
 1.2289436008118710861619676258142970506111002000238983779491510e-02,
-9.5777978992357099981473097037135186082832338827934897334916420e-03,
-5.0859916492334298817976365835789211946753938077611545497335470e-03,
 6.1377545867405210895968018836319212211457125450425199876412340e-03,
 1.4280887940707621073555858706698421326091590406258950900701110e-03,
-3.3576443809223832295677325652986656390373485859611270755079370e-03,
 7.6159694351727365467696499238953174515347030660161162573001600e-06,
 1.5496374697023629755617192465397877172044386379978249357876880e-03,
-3.3466921642508549616085261215245969080411099183613062822013100e-04,
-5.8648103189918175321758092241314567383671010356941882234088410e-04,
 2.6483288199612890393028101226997109660485653680475752186931340e-04,
 1.7000122836612490435846901947167677712042077426257463085229350e-04,
-1.3658830722611616025599267147447464225675091774435940457096530e-04,
-2.9769959628485097439442258664885196685852426559806566465443190e-05,
 5.3041431229133102225383179806863746960056055334756855874866830e-05,
-2.4370015268277898609904294785405567526943896934326688310737690e-06,
-1.5724420772702816936632889664058612156928059727379819861214470e-05,
 4.3080478617167311913504934379375132207374504101328780321631790e-06,
 3.3533458628713098893908771680461336573771056816187083552666880e-06,
-1.8959296176931532884938910518754444397533185481059981665745350e-06,
-3.9039317332873061666575194684945119207607673883978257753267450e-07,
 5.3023686169047609170743526339157432507696006358292296008125200e-07,
-3.7003083782051245379864026449188791498940359101064890825123640e-08,
-9.9903969445349007557817284775612407621914434223182491288667400e-08,
 3.0081886507190669282302689186617182745049550450225502170513010e-08,
 1.0849027337899348252665602401004498847027493033265717473230860e-08,
-7.4581165528930376311924076112627885935059886383658404093671170e-09,
 5.8979513103843615754703558611620225011724919378377129698656190e-11,
 1.0308233454854333838117004814885574220052101680691637797309080e-09,
-2.4335455737516729361688772504059408172273679372302898012516480e-10,
-6.4079382565018890184306083232359744062191931769182846649737270e-11,
 4.0005366272537445107427882013540930064717104166710022443025860e-11,
-3.1256393571085575405980982286781507685281215653913762656272940e-12,
-2.5670654761550814492046438524284015302835196856382560747528500e-12,
 8.0150885336879009219486054187893248261156164163433910812889790e-13,
-2.5979543288938480843151982050943891457066801292089986388029950e-14,
-3.3977208567962674319567838256590695969403351301008719123295560e-14,
 8.6240374347200892026803376636927776828107146500608058324061350e-15,
-9.2980125293241854209215556647198635018483150991167251843703390e-16,
 4.0146287123334886543185691646142203080460210911781846542509820e-17};

const double Daub36[72] = {
 2.8679251827559463346304794730292386155355117758942627110547050e-06,
 6.8260286785463586917486291022096053622403442665050359817917150e-05,
 7.6021510996684882858697926771060821001412750548923893791985450e-04,
 5.2402973774098843662016035243929956960421749371944352350039410e-03,
 2.4890565644827964848859273339591155794030233470447297392552550e-02,
 8.5652092595264090838647169955211114864375947503778565247727040e-02,
 2.1775695309790081496379459157199997462489697056506255334158760e-01,
 4.0643369770825534674077939902503844459031516307685581421253820e-01,
 5.3226689526072869147774447486414620272135547231539069011293370e-01,
 4.1787533560096978636206345593742364552222753029969311782659190e-01,
 4.3975197529348629938621828983587637831107455592389821796901320e-02,
-2.9442103958911457111007159698987589407224588873778446334436750e-01,
-2.4680703697812552705247982786226984465665207182303138890860160e-01,
 9.8114204163114770505184013714015680389434373222999135140497280e-02,
 2.4653727760897421105297091118095954346564187628981527066213560e-01,
 7.2785150957922290096876822994603828786431390266689588844296410e-03,
-1.9933720560864961986033634000947841427141622567921825705410360e-01,
-4.5861400746392716391451262228774831743002971373998329604573940e-02,
 1.5410623662764288417763163004206548758838428194136233953582620e-01,
 5.0276180073538428620368169728098840967617060360197483168909130e-02,
-1.1880375431013563168018169313835474460731529510444442244495010e-01,
-3.9880853575513175840916999679240440341003742570758642609341020e-02,
 9.1156782258016544063360592813067151510589030553705220318437710e-02,
 2.5038721449568489899194842967098468605691809930403836219805460e-02,
-6.8209016636817511248804363442655386905803581087145407631251190e-02,
-1.1319100316817427943818080821736950221230562808216113545778830e-02,
 4.8513083547809085386162676623157356322929897490132612070463670e-02,
 1.4249726617653916031478026073785423963234296576600097556524040e-03,
-3.1980720677639696544702935137423446011727396882742516418737780e-02,
 3.9840401987170048573971794867900823213142913666561512134290680e-03,
 1.9063594780625359328775761643681982748581085136968327288892090e-02,
```

```
-5.657813245058818380424016973516714570499161434975761798379020e-03,
-9.990263473281372348001743806489172665465685056975652497503772e-03,
 5.022989106665829004699819220796538830393945994687289792465541e-03,
 4.413484835350575251918616780287775585471012556848037301025999e-03,
-3.484541445404883311209541395428535732697661971818727286003028e-03,
-1.503074066296643749549363655363411879858070202740814054964603e-03,
 1.990793771851737270404293245701878186600899439513475823305914e-03,
 2.776812795712026068152384207605140383490242756921936501940389e-04,
-9.463403823261101964604918059447913047725482130063492242779878e-04,
 8.614565758992702032613879159402330909634737204578606399403107e-05,
 3.693507284967510502620040341882236687749563414433432842567511e-04,
-1.155118895843527096848376999413102395191976350936666573818799e-04,
-1.131899468084665671727391922924411467938450743565106978099456e-04,
 6.694741196930590257104231749283786251555566773398199990337698e-05,
 2.375106683660860777161950832380341362257503761490580896617678e-05,
-2.731390824654337912922346414722045404779935825834384250023192e-05,
-1.183471059985615942783182762352360917304348034947412986608322e-06,
 8.372218198160788432628056043217491552198857358432112275253310e-06,
-1.586145782434577495502614631566211839722879492827911790709498e-06,
-1.870811602859180713762972281154953528056257451900381097476968e-06,
 8.311421279707778528163597405935375886855029592150424544500718e-07,
 2.548423522556577831218519052844387478819866531902854523544709e-07,
-2.455377658434232699135878286794578515387138194247693201846263e-07,
 2.753249073339512254085076456700241929492720457889076058451072e-09,
 4.799043465450992009934526867650497683545716858606119786327559e-08,
-1.156093688817008406756913949175208452083765368825442482226093e-08,
-5.612784343327791397474114357094368557982413895802980814813369e-09,
 3.138841695782424018351567952158415003571380699236147752239001e-09,
 1.090815553713751810964713058800448676068475673611349566405716e-10,
-4.512545778563249634425200856088490195004077806062978067796020e-10,
 8.962418203859611987065968320295929679774693465791367610044773e-11,
 3.037429098112535221800013609576297196061786927734556635696416e-11,
-1.599716689261357143200396922409448515398648489795044468046420e-11,
 8.876846287217374213524399682895564055949886050748321818411161e-13,
 1.070969357114017002424433471621197579059927261727846375968378e-12,
-3.029285026974877268896134589769473854669758797446795757329862e-13,
 5.542263182639804235231685861028995158694397223907295269180336e-15,
 1.338071386299105896025578761458472955294763310766371178363783e-14,
-3.204628543401749860439316638848579711789176444320134355253750e-15,
 3.339971984818693213132578777712503670014459411167839211495237e-16,
-1.403274175373190617489823209168013922564353495443487431242610e-17};

const double Daub37[74] = {
 2.022060862498392121815038335333633351464174415618614893795880e-06,
 4.942343750628132004714286117434454499485737947791397867195910e-05,
 5.662418377066724013768394373249439163518654840493603575144737e-04,
 4.024140368257286770702140124893772447952256842478891548092703e-03,
 1.976228615387959153244055502205017461538589475705618414896893e-02,
 7.058482597718160832030361890793007659963483925312132741868671e-02,
 1.873263318620649448028843491747601576761901656888288838192023e-01,
 3.684409724003061409445838616964941132670287724754729425204047e-01,
 5.181670408556228873104519667534437205387109579265718071174178e-01,
 4.622075536616057145505448401528172070050768534504278694229363e-01,
 1.308789632330201726057701201017649601034381070893275586898075e-01,
-2.461804297610834132869018581145720710365433914584680691693717e-01,
-2.943759152626617722808219575932673733674290772235644691367427e-01,
 1.967150045235938977077768648740052380288156507222647187301894e-02,
 2.515232543602686933435224095078166291442923992611593827552710e-01,
 8.180602838721862339029076982652411696000045533716726027662147e-02,
-1.819622917786080007408824256525225216444443143868752611284260e-01,
-1.084517138233017845554078812341876568514835176341639783558543e-01,
 1.299296469598537527842528895259188653120602318620944502979726e-01,
 1.017802968388141797470948228505865617480048287983176581607964e-01,
-9.660754061668439030915405045955772715988585374771282291315496e-02,
-8.233021190655740867404073660920379414988302492018783774702028e-02,
 7.504761994836017933579005072594245435071674452882148228583865e-02,
 5.956741087152995245435589042520108066877114768216272503684398e-02,
-5.925681563265897095153806724965924334077555174281436189512239e-02,
-3.825382947938424882011108885090442116802994193611884738133373e-02,
 4.580794415126833246633256156110381805848138158784734496981778e-02,
 2.097280059259754883313769469036393294461497749083921162354229e-02,
-3.352358406410096994358662875913243067234786296009238949920582e-02,
-8.833493890410232394064187990625563257107429109130726291528648e-03,
 2.261865154459947356571431658958802912061105608212828675323452e-02,
 1.690472383484423743663952859090705636512807161536954018400081e-03,
-1.376398196289478433857985486097070339786225136728067000591187e-02,
 1.519305778833399218481261844599507408563295102235964076544334e-03,
 7.387757452855583640107787619408806919082115520707105052944171e-03,
-2.248053187003824706127276829147166466869908326245810952521710e-03,
-3.394523276408398601988475786247462646314228994098320665709345e-03,
 1.816871343801423525477184531347879515909226877688306010517914e-03,
 1.263934258117477182626760951047019242187910977671449470318766e-03,
-1.111484865318630197259018233162929628309920117691177260742614e-03,
```

```
-3.280788470880198419407186455190899535706232295554613820907245e-04,
 5.490532773373631230219769273898345809368332716288071475378651e-04,
 1.534439023195503211083338679106161291342621676983096723309776e-05,
-2.208944032455493852493630802748509781675182699536797043565515e-04,
 4.336726125945695214852398433524024058216834313839357806404424e-05,
 7.055138782065465075838703109997365141906130284669094131032488e-05,
-3.098662927619930052417611453170793938796310141219293329658062e-05,
-1.639162496160583099236044020495877311072716199713679670940295e-05,
 1.354327718416781810683349121150634031343717637827354228989989e-05,
 1.849945003115590390789683032647334516600314304175482456338006e-06,
-4.309941556597092389020622638271988877959028012481278949268461e-06,
 4.854731396996411681769911684430785681028852413859386141424939e-07,
 1.002121399297177629772998172241869405763288457224082581829033e-06,
-3.494948603445727645895194867933547164628229076947330682199174e-07,
-1.509885388671583553484927666148474078148724554849968758642331e-07,
 1.109031232216439389999036327867142640916239658806376290861690e-07,
 5.350657515461434290618742656970344024396382191417247602674540e-09,
-2.252193836724805775389816424695618411834716065179297102428180e-08,
 4.224485706362419268050011630338101126995607958955688879525896e-09,
 2.793974465953982659829387370821677112004867350709951380622807e-09,
-1.297205001469435139867686007585972538983682739297235604327668e-09,
-1.031411129096974965677950646498153071722880698222864687038596e-10,
 1.946164894082315021308714557636277980079559327508927751052218e-10,
-3.203398244123241367987902201268363088933939831689591684670080e-11,
-1.398415715537641487959551682557483348661602836709278513081908e-11,
 6.334955440973913249611879065201632922100533284261000819747915e-12,
-2.096363194234800541614775742755557132795493812648810308432580e-13,
-4.421612409872105367333572734854401373201808896976552663098518e-13,
 1.138052830921439682522395208295427884729893377395129205716662e-13,
-4.518889607463726394454509623712773172513778367070839294449849e-16,
-5.243025691884205832260354503748325334301994904062750850180233e-15,
 1.189012387508252879928637969242590755033933791160383262132698e-15,
-1.199280335852879554967035114674445327319437557227036460257649e-16,
 4.906615064935203694857690087429901193139905690549533773201453e-18};

const double Daub38[76] = {
 1.425776641674131672055420247567865803211784397464191115245081e-06,
 3.576251994264023012742569014888876217958307227940126418281357e-05,
 4.211702664727116432247014444906469155300573201130549739553848e-04,
 3.083088119253751774288740090262741910177322520624582862578292e-03,
 1.563724934757215617277490102724080070486270026632620664785632e-02,
 5.788994361285925649727664279317241952513246287766481213301801e-02,
 1.600719935641106973482800861166599685169395465055048951307626e-01,
 3.307757814110146511493637534404611754800768677041577030757306e-01,
 4.965911753117180976599171147718708939352414838951726087564419e-01,
 4.933560785171007975728485346997317064969513623594359091115804e-01,
 2.130505713555785138286743353458562451255624665951160445122307e-01,
-1.828676677083358907975548507946239135218223185041410632924815e-01,
-3.216756378089978628483471725406916361929841940528189059002548e-01,
-6.226650604782432226643360160478765847565862101045597180310490e-02,
 2.321259638353531085028708104285994998671615563662858079262996e-01,
 1.499851196187170199586403453788927307298226028262603028635758e-01,
-1.417956859730596216710053144522330276392591055375830654519080e-01,
-1.599125651582443618288533214523534937804208844386102639177693e-01,
 8.563812155615105741612217814369165313487129645536001850276987e-02,
 1.414147340733826800884683119379170594092606174915755283496153e-01,
-5.658645863072738145681787657843320646815509410635114234947902e-02,
-1.147311707107443752394144019458942779715665489230169950201022e-01,
 4.309589543304764288137871223616030624246568683595408792078602e-02,
 8.720439826203975011910714164154456762073786124233088471855868e-02,
-3.660510340287429567372071039506772372567938710943432838908247e-02,
-6.176620870841315993604736705613246241897497782373337911398117e-02,
 3.198987753153780630818381136366859026137035450576631134176875e-02,
 4.005498110511594820952087086241114309038577379366732959648548e-02,
-2.689149388089451438550851767715967313417890393287236700072071e-02,
-2.311413402054931680856913553585621248925303865540203357180768e-02,
 2.090464525565524340215982365351342094670261491526831672682244e-02,
 1.129049727868596484270081487761544232851115891449843967151657e-02,
-1.470188206539868213708986402816605045648481224662435114088245e-02,
-4.131306656031089274123231103326745723188134548520938157995702e-03,
 9.214785032197180512031534870181734003522861645903894504302286e-03,
 5.625715748403532005741565594881148757066703437214522101740941e-04,
-5.071314509218348093935061417505663002006821323958752649640329e-03,
 7.169821821064019257784165364894915621888541496773370435889585e-04,
 2.400697781890973183892306914082592143984140550210130139535193e-03,
-8.448626665537775009068937851465856973251363010924003314643612e-04,
-9.424614077227377964015942271780098283910230639908018778588910e-04,
 5.810759750532863662020321063678196633409555706981476723988312e-04,
 2.817639250380670746018048967535608190123523180612961062603672e-04,
-3.031020460726611993600629020329784682496477106470427787747855e-04,
-4.555682696668420274688683005987764360677217149927938344795290e-05,
 1.262043350166170705382346537131817701361522387904917335958705e-04,
-1.155409103833717192628479047983460953381959342642374175822863e-05,
```

```
-4.1751416485403977972963250657757113091974119262894124682808 01e-05,
```

DFT (click this to go back to the index)

Type : fourier transform

References : Posted by Andy Mucho

Code :
```
AnalyseWaveform(float *waveform, int framesize)
{
   float aa[MaxPartials];
   float bb[MaxPartials];
   for(int i=0;i<partials;i++)
   {
     aa[i]=0;
     bb[i]=0;
   }

   int hfs=framesize/2;
   float pd=pi/hfs;
   for (i=0;i<framesize;i++)
   {
     float w=waveform[i];
     int im = i-hfs;
     for(int h=0;h<partials;h++)
     {
        float th=(pd*(h+1))*im;
        aa[h]+=w*cos(th);
        bb[h]+=w*sin(th);
     }
   }
   for (int h=0;h<partials;h++)
       amp[h]= sqrt(aa[h]*aa[h]+bb[h]*bb[h])/hfs;
}
```

Envelope detector (click this to go back to the index)

References : Posted by Bram

Notes :
Basicaly a one-pole LP filter with different coefficients for attack and release fed by the abs() of the signal. If you don't need different attack and decay settings, just use in->abs()->LP

Code :
```
//attack and release in milliseconds
float ga = (float) exp(-1/(SampleRate*attack));
float gr = (float) exp(-1/(SampleRate*release));

float envelope=0;

for(...)
{
  //get your data into 'input'
  EnvIn = abs(input);

  if(envelope < EnvIn)
  {
     envelope *= ga;
     envelope += (1-ga)*EnvIn;
  }
  else
  {
     envelope *= gr;
     envelope += (1-gr)*EnvIn;
  }
  //envelope now contains.........the envelope ;)
}
```

Comments
from : arguru@smartelectronix
comment : Nice , just a typo: //attack and release is entered in SECONDS actually in this code ;)

Envelope follower with different attack and release (click this to go back to the index)

References : Posted by Bram

Notes :
xxxx_in_ms is xxxx in milliseconds ;-)

Code :
```
init::

attack_coef = exp(log(0.01)/( attack_in_ms * samplerate * 0.001));
release_coef = exp(log(0.01)/( release_in_ms * samplerate * 0.001));
envelope = 0.0;

loop::

tmp = fabs(in);
if(tmp > envelope)
    envelope = attack_coef * (envelope - tmp) + tmp;
else
    envelope = release_coef * (envelope - tmp) + tmp;
```

Comments
from : jm@kampsax.dtu.dk
comment : the expressions of the form:

xxxx_coef = exp(log(0.01)/(xxxx_in_ms * samplerate * 0.001));

can be simplified a little bit to:

xxxx_coef = pow(0.01, 1.0/(xxxx_in_ms * samplerate * 0.001));

from : kainhart@hotmail.com
comment : Excuse me if I'm asking a lame question but is the envelope variable the output for the given input sample? Also would this algorithm apply to each channel independently for a stereo signal? One more question what is an Envelope Follower, what does it sound like?

from : yanyuqiang@hotmail.com
comment : What's the difference between this one and the one you posted named 'Envelope detector'? Different definiton? What's the exact definition of release time and attack time?

Fast in-place Walsh-Hadamard Transform (click this to go back to the index)

Type : wavelet transform

References : Posted by Timo H Tossavainen

Notes :
IIRC, They're also called walsh-hadamard transforms.
Basically like Fourier, but the basis functions are squarewaves with different sequencies.
I did this for a transform data compression study a while back.
Here's some code to do a walsh hadamard transform on long ints in-place (you need to divide by n to get transform) the order is bit-reversed at output, IIRC.
The inverse transform is the same as the forward transform (expects bit-reversed input). i.e. x = 1/n * FWHT(FWHT(x)) (x is a vector)

Code :
```
void inline wht_bfly (long& a, long& b)
{
        long tmp = a;
        a += b;
        b = tmp - b;
}

// just a integer log2
int inline l2 (long x)
{
        int l2;
        for (l2 = 0; x > 0; x >>=1)
        {
                ++ l2;
        }

        return (l2);
}

////////////////////////////////////////////
// Fast in-place Walsh-Hadamard Transform //
////////////////////////////////////////////

void FWHT (std::vector& data)
{
  const int log2 = l2 (data.size()) - 1;
  for (int i = 0; i < log2; ++i)
  {
    for (int j = 0; j < (1 << log2); j += 1 << (i+1))
    {
       for (int k = 0; k < (1<<i); ++k)
       {
           wht_bfly (data [j + k], data [j + k + (1<<i)]);
       }
    }
  }
}
```

Comments
from : i_arslan@gmx.net
comment : How can i implement this code in matlab ?

from : Noor98z@hotmail.com
comment : Need an Implemtation of Walsh Transform Matlab Code

from : ambadarneh@hotmail.com
comment : well sir nice work ,but how can i get the code in MATLAB .
could you help me please.

thank you
yours sicerly
Ala' Badarneh
Jordan University of Science & Technology
Biomedical Engineering Department

FFT (click this to go back to the index)

References : Toth Laszlo

Linked file : rvfft.ps

Linked file : rvfft.cpp (this linked file is included below)

Notes :
A paper (postscript) and some C++ source for 4 different fft algorithms, compiled by Toth Laszlo from the Hungarian Academy of Sciences Research Group on Artificial Intelligence.
Toth says: "I've found that Sorensen's split-radix algorithm was the fastest, so I use this since then (this means that you may as well delete the other routines in my source - if you believe my results)."

Linked files

```
//
//                  FFT library
//
//  (one-dimensional complex and real FFTs for array
//  lengths of 2^n)
//
// Author: Toth Laszlo (tothl@inf.u-szeged.hu)
//
// Research Group on Artificial Intelligence
//  H-6720 Szeged, Aradi vertanuk tere 1, Hungary
//
// Last modified: 97.05.29
/////////////////////////////////////////////////////////

#include <math.h>
#include <stdlib.h>
#include "pi.h"

/////////////////////////////////////////////////////////
//Gives back "i" bit-reversed where "size" is the array
//length
//currently none of the routines call it

long bitreverse(long i, long size){

 long result,mask;

 result=0;
 for(;size>1;size>>=1){
  mask=i&1;
  i>>=1;
  result<<=1;
  result|=mask;
 }

/*     __asm{       the same in assebly
     mov eax,i
  mov ecx,sizze
     mov ebx,0
   l:shr eax,1
     rcl ebx,1
  shr ecx,1
  cmp ecx,1
     jnz l
     mov result,ebx
     }*/
  return result;
}

////////////////////////////////////////////////////////
//Bit-reverser for the Bruun FFT
//Parameters as of "bitreverse()"

long bruun_reverse(long i, long sizze){

 long result, add;

 result=0;
 add=sizze;

 while(1){
 if(i!=0) {
  while((i&1)==0) { i>>=1; add>>=1;}
  i>>=1; add>>=1;
  result+=add;
 }
 else {result<<=1;result+=add; return result;}
```

```
 if(i!=0) {
  while((i&1)==0) { i>>=1; add>>=1;}
  i>>=1; add>>=1;
  result-=add;
 }
 else {result<<=1;result-=add; return result;}
 }
}

/*assembly version
long bruun_reverse(long i, long sizze){

 long result;

 result=0;

 __asm{
  mov edx,sizze
  mov eax,i
  mov ebx,0
  l: bsf cx,eax
  jz kesz1
  inc cx
  shr edx,cl
     add ebx,edx
  shr eax,cl
  bsf cx,eax
  jz kesz2
  inc cx
  shr edx,cl
  sub ebx,edx
  shr eax,cl
  jmp l
kesz1:
  shl ebx,1

  add ebx,edx
  jmp vege
kesz2:
  shl ebx,1
  sub ebx,edx

  vege: mov result,ebx
 }
 return result;
}*/

/////////////////////////////////////////////////////////
// Decimation-in-freq radix-2 in-place butterfly
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// suggested use:
// intput in normal order
// output in bit-reversed order
//
// Source: Rabiner-Gold: Theory and Application of DSP,
// Prentice Hall,1978

void dif_butterfly(double *data,long size){

 long angle,astep,dl;
 double xr,yr,xi,yi,wr,wi,dr,di,ang;
 double *l1, *l2, *end, *ol2;

     astep=1;
  end=data+size+size;
     for(dl=size;dl>1;dl>>=1,astep+=astep){
       l1=data;
       l2=data+dl;
                    for(;l2<end;l1=l2,l2=l2+dl){
       ol2=l2;
                          for(angle=0;l1<ol2;l1+=2,l2+=2){
                             ang=2*pi*angle/size;
                             wr=cos(ang);
                             wi=-sin(ang);
                             xr=*l1+*l2;
                             xi=*(l1+1)+*(l2+1);
                             dr=*l1-*l2;
                             di=*(l1+1)-*(l2+1);
                             yr=dr*wr-di*wi;
                             yi=dr*wi+di*wr;
```

```
                                *(l1)=xr;*(l1+1)=xi;
                                *(l2)=yr;*(l2+1)=yi;
                                angle+=astep;
          }
       }
  }
}

/////////////////////////////////////////////////////////
// Decimation-in-time radix-2 in-place inverse butterfly
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// suggested use:
// intput in bit-reversed order
// output in normal order
//
// Source: Rabiner-Gold: Theory and Application of DSP,
// Prentice Hall,1978

void inverse_dit_butterfly(double *data,long size){

 long angle,astep,dl;
 double xr,yr,xi,yi,wr,wi,dr,di,ang;
 double *l1, *l2, *end, *ol2;

     astep=size>>1;
  end=data+size+size;
     for(dl=2;astep>0;dl+=dl,astep>>=1){
       l1=data;
       l2=data+dl;
                      for(;l2<end;l1=l2,l2=l2+dl){
       ol2=l2;
                            for(angle=0;l1<ol2;l1+=2,l2+=2){
                               ang=2*pi*angle/size;
                               wr=cos(ang);
                               wi=sin(ang);
                               xr=*l1;
                               xi=*(l1+1);
                               yr=*l2;
                               yi=*(l2+1);
                               dr=yr*wr-yi*wi;
                               di=yr*wi+yi*wr;
                               *(l1)=xr+dr;*(l1+1)=xi+di;
                               *(l2)=xr-dr;*(l2+1)=xi-di;
                               angle+=astep;
          }
       }
  }
}

/////////////////////////////////////////////////////////
// data shuffling into bit-reversed order
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// Source: Rabiner-Gold: Theory and Application of DSP,
// Prentice Hall,1978

void unshuffle(double *data, long size){

 long i,j,k,l,m;
 double re,im;

 //old version - turned out to be a bit slower
    /*for(i=0;i<size-1;i++){
             j=bitreverse(i,size);
             if (j>i){   //swap
                    re=data[i+i];im=data[i+i+1];
                    data[i+i]=data[j+j];data[i+i+1]=data[j+j+1];
                    data[j+j]=re;data[j+j+1]=im;
    }
 }*/

l=size-1;
m=size>>1;
for (i=0,j=0; i<l ; i++){
   if (i<j){
    re=data[j+j]; im=data[j+j+1];
    data[j+j]=data[i+i]; data[j+j+1]=data[i+i+1];
    data[i+i]=re; data[i+i+1]=im;
```

```
    }
   k=m;
   while (k<=j){
    j-=k;
    k>>=1;
    }
   j+=k;
      }
}

/////////////////////////////////////////////////////////
// used by realfft
// parameters as above
//
// Source: Brigham: The Fast Fourier Transform
// Prentice Hall, ?

void realize(double *data, long size){

 double xr,yr,xi,yi,wr,wi,dr,di,ang,astep;
 double *l1, *l2;

 l1=data;
 l2=data+size+size-2;
    xr=*l1;
    xi=*(l1+1);
    *l1=xr+xi;
    *(l1+1)=xr-xi;
 l1+=2;
 astep=pi/size;
    for(ang=astep;l1<=l2;l1+=2,l2-=2,ang+=astep){
                               xr=(*l1+*l2)/2;
                               yi=(-(*l1)+(*l2))/2;
                               yr=(*(l1+1)+*(l2+1))/2;
                               xi=(*(l1+1)-*(l2+1))/2;
                               wr=cos(ang);
                               wi=-sin(ang);
                               dr=yr*wr-yi*wi;
                               di=yr*wi+yi*wr;
                               *l1=xr+dr;
                               *(l1+1)=xi+di;
                               *l2=xr-dr;
                               *(l2+1)=-xi+di;
 }
}

/////////////////////////////////////////////////////////
// used by inverse realfft
// parameters as above
//
// Source: Brigham: The Fast Fourier Transform
// Prentice Hall, ?

void unrealize(double *data, long size){

 double xr,yr,xi,yi,wr,wi,dr,di,ang,astep;
 double *l1, *l2;

 l1=data;
 l2=data+size+size-2;
    xr=(*l1)/2;
    xi=(*(l1+1))/2;
    *l1=xr+xi;
    *(l1+1)=xr-xi;
 l1+=2;
 astep=pi/size;
    for(ang=astep;l1<=l2;l1+=2,l2-=2,ang+=astep){
                               xr=(*l1+*l2)/2;
                               yi=-(-(*l1)+(*l2))/2;
                               yr=(*(l1+1)+*(l2+1))/2;
                               xi=(*(l1+1)-*(l2+1))/2;
                               wr=cos(ang);
                               wi=-sin(ang);
                               dr=yr*wr-yi*wi;
                               di=yr*wi+yi*wr;
                               *l2=xr+dr;
                               *(l1+1)=xi+di;
                               *l1=xr-dr;
                               *(l2+1)=-xi+di;
 }
}

/////////////////////////////////////////////////////////
```

```
// in-place Radix-2 FFT for complex values
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// output is in similar order, normalized by array length
//
// Source: see the routines it calls ...

void fft(double *data, long size){

 double *l, *end;

 dif_butterfly(data,size);
 unshuffle(data,size);

 end=data+size+size;
 for(l=data;l<end;l++){*l=*l/size;};
}


/////////////////////////////////////////////////////////
// in-place Radix-2 inverse FFT for complex values
// data: array of doubles:
// re(0),im(0),re(1),im(1),...,re(size-1),im(size-1)
// it means that size=array_length/2 !!
//
// output is in similar order, NOT normalized by
// array length
//
// Source: see the routines it calls ...

void ifft(double* data, long size){

 unshuffle(data,size);
 inverse_dit_butterfly(data,size);
}

/////////////////////////////////////////////////////////
// in-place Radix-2 FFT for real values
// (by the so-called "packing method")
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source: see the routines it calls ...

void realfft_packed(double *data, long size){

 double *l, *end;
 double div;

 size>>=1;
 dif_butterfly(data,size);
 unshuffle(data,size);
 realize(data,size);

 end=data+size+size;
 div=size+size;
 for(l=data;l<end;l++){*l=*l/div;};
}


/////////////////////////////////////////////////////////
// in-place Radix-2 inverse FFT for real values
// (by the so-called "packing method")
// data: array of doubles:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
//
// output:
// re(0),re(1),re(2),...,re(size-1)
// NOT normalized by array length
//
//Source: see the routines it calls ...

void irealfft_packed(double *data, long size){

 double *l, *end;

 size>>=1;
```

```
 unrealize(data,size);
 unshuffle(data,size);
 inverse_dit_butterfly(data,size);

 end=data+size+size;
 for(l=data;l<end;l++){*l=(*l)*2;};
}


/////////////////////////////////////////////////////////
// Bruun FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),...,re(i),im(i)... pairs in
// "bruun-reversed" order
// normalized by array length
//
// Source:
// Bruun: z-Transform DFT Filters and FFT's
// IEEE Trans. ASSP, ASSP-26, No. 1, February 1978
//
// Comments:
// (this version is implemented in a manner that every
// cosine is calculated only once;
// faster than the other version (see next)

void realfft_bruun(double *data, long size){

 double *end, *l0, *l1, *l2, *l3;
 long dl, dl2, dl_o, dl2_o, i, j, k, kk;
 double d0,d1,d2,d3,c,c2,p4;

 end=data+size;
 //first filterings, when there're only two taps
 size>>=1;
 dl=size;
 dl2=dl/2;
 for(;dl>1;dl>>=1,dl2>>=1){
  l0=data;
  l3=data+dl;
  for(i=0;i<dl;i++){
   d0=*l0;
   d2=*l3;
   *l0=d0+d2;
   *l3=d0-d2;
   l0++;
   l3++;
  }
 }
 l0=data;l1=data+1;
 d0=*l0;d1=*l1;
 *l0=d0+d1;*l1=d0-d1;
 l1+=2;
 *l1=-(*l1);

 //the remaining filterings

 p4=pi/(2*size);
 j=1;
 kk=1;
 dl_o=size/2;
 dl2_o=size/4;
 while(dl_o>1){
 for(k=0;k<kk;k++){
 c2=p4*bruun_reverse(j,size);
 c=2*cos(c2);
 c2=2*sin(c2);
 dl=dl_o;
 dl2=dl2_o;
 for(;dl>1;dl>>=1,dl2>>=1){
   l0=data+((dl*j)<<1);
   l1=l0+dl2;l2=l0+dl;l3=l1+dl;
   for(i=0;i<dl2;i++){
    d1=(*l1)*c;
    d2=(*l2)*c;
    d3=*l3+(*l1);
    d0=*l0+(*l2);
    *l0=d0+d1;
    *l1=d3+d2;
    *l2=d0-d1;
    *l3=d3-d2;
```

```
    l0++;
    l1++;
    l2++;
    l3++;
   }
  }
  //real conversion
  l3-=4;
  *l3=*l3-c*(*l0)/2;
  *l0=-c2*(*l0)/2;
  *l1=*l1+c*(*l2)/2;
  *l2=-c2*(*l2)/2;
 j++;
 }
 dl_o>>=1;
 dl2_o>>=1;
 kk<<=1;
 }

 //division with array length
 size<<=1;
 for(i=0;i<size;i++) data[i]=data[i]/size;
}

/////////////////////////////////////////////////////////
// Bruun FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source: see the routines it calls ...
void realfft_bruun_unshuffled(double *data, long size){

 double *data2;
 long i,j,k;

 realfft_bruun(data,size);
 //unshuffling - cannot be done in-place (?)
 data2=(double *)malloc(size*sizeof(double));
 for(i=1,k=size>>1;i<k;i++){
  j=bruun_reverse(i,k);
  data2[j+j]=data[i+i];
  data2[j+j+1]=data[i+i+1];
 }
 for(i=2;i<size;i++) data[i]=data2[i];
 free(data2);
}


/////////////////////////////////////////////////////////
// Bruun FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),...,re(i),im(i)... pairs in
// "bruun-reversed" order
// normalized by array length
//
// Source:
// Bruun: z-Transform DFT Filters and FFT's
// IEEE Trans. ASSP, ASSP-26, No. 1, February 1978
//
// Comments:
// (this version is implemented in a row-by-row manner;
// control structure is simpler, but there are too
// much cosine calls - with a cosine lookup table
// probably this would be slightly faster than bruun_fft

/*void realfft_bruun2(double *data, long size){

 double *end, *l0, *l1, *l2, *l3;
 long dl, dl2, i, j;
 double d0,d1,d2,d3,c,c2,p4;

 end=data+size;
 p4=pi/(size);
 size>>=1;
 dl=size;
 dl2=dl/2;
```

```
 //first filtering, when there're only two taps
 for(;dl>1;dl>>=1,dl2>>=1){
  l0=data;
  l3=data+dl;
  for(i=0;i<dl;i++){
   d0=*l0;
   d2=*l3;
   *l0=d0+d2;
   *l3=d0-d2;
   l0++;
   l3++;
  }
  //the remaining filterings
  j=1;
  while(l3<end){
   l0=l3;l1=l0+dl2;l2=l0+dl;l3=l1+dl;
   c=2*cos(p4*bruun_reverse(j,size));
   for(i=0;i<dl2;i++){
    d0=*l0;
    d1=*l1;
    d2=*l2;
    d3=*l3;
    *l0=d0+c*d1+d2;
    *l2=d0-c*d1+d2;
    *l1=d1+c*d2+d3;
    *l3=d1-c*d2+d3;
    l0++;
    l1++;
    l2++;
    l3++;
   }
   j++;
  }
 }

 //the last row: transform of real data
 //the first two cells
 l0=data;l1=data+1;
 d0=*l0;d1=*l1;
 *l0=d0+d1;*l1=d0-d1;
 l1+=2;
 *l1=-(*l1);
 l0+=4;l1+=2;
 //the remaining cells
 j=1;
 while(l0<end){
   c=p4*bruun_reverse(j,size);
   c2=sin(c);
   c=cos(c);
   *l0=*l0-c*(*l1);
   *l1=-c2*(*l1);
   l0+=2;
   l1+=2;
   *l0=*l0+c*(*l1);
   *l1=-c2*(*l1);
   l0+=2;
   l1+=2;
   j++;
   }
 //division with array length
 for(i=0;i<size;i++) data[i]=data[i]/size;
}
*/

//the same as realfft_bruun_unshuffled,
//but calls realfft_bruun2
/*void realfft_bruun_unshuffled2(double *data, long size){

 double *data2;
 long i,j,k;

 realfft_bruun2(data,size);
 //unshuffling - cannot be done in-place (?)
 data2=(double *)malloc(size*sizeof(double));
 for(i=1,k=size>>1;i<k;i++){
  j=bruun_reverse(i,k);
  data2[j+j]=data[i+i];
  data2[j+j+1]=data[i+i+1];
 }
 for(i=2;i<size;i++) data[i]=data2[i];
 free(data2);
}*/
```

```
/////////////////////////////////////////////////////////
// Sorensen in-place split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(1),re(2),...,re(size/2),im(size/2-1),...,im(1)
// normalized by array length
//
// Source:
// Sorensen et al: Real-Valued Fast Fourier Transform Algorithms,
// IEEE Trans. ASSP, ASSP-35, No. 6, June 1987

void realfft_split(double *data,long n){

  long i,j,k,i5,i6,i7,i8,i0,id,i1,i2,i3,i4,n2,n4,n8;
  double t1,t2,t3,t4,t5,t6,a3,ss1,ss3,cc1,cc3,a,e,sqrt2;

  sqrt2=sqrt(2.0);
  n4=n-1;

  //data shuffling
      for (i=0,j=0,n2=n/2; i<n4 ; i++){
   if (i<j){
    t1=data[j];
    data[j]=data[i];
    data[i]=t1;
    }
   k=n2;
   while (k<=j){
    j-=k;
    k>>=1;
    }
   j+=k;
      }

/*-----------------------*/

 //length two butterflies
 i0=0;
 id=4;
   do{
       for (; i0<n4; i0+=id){
   i1=i0+1;
   t1=data[i0];
   data[i0]=t1+data[i1];
   data[i1]=t1-data[i1];
  }
    id<<=1;
       i0=id-2;
       id<<=1;
    } while ( i0<n4 );

   /*-----------------------*/
   //L shaped butterflies
n2=2;
for(k=n;k>2;k>>=1){
 n2<<=1;
 n4=n2>>2;
 n8=n2>>3;
 e = 2*pi/(n2);
 i1=0;
 id=n2<<1;
 do{
     for (; i1<n; i1+=id){
   i2=i1+n4;
   i3=i2+n4;
   i4=i3+n4;
   t1=data[i4]+data[i3];
   data[i4]-=data[i3];
   data[i3]=data[i1]-t1;
   data[i1]+=t1;
   if (n4!=1){
    i0=i1+n8;
    i2+=n8;
    i3+=n8;
    i4+=n8;
    t1=(data[i3]+data[i4])/sqrt2;
    t2=(data[i3]-data[i4])/sqrt2;
    data[i4]=data[i2]-t1;
    data[i3]=-data[i2]-t1;
    data[i2]=data[i0]-t2;
    data[i0]+=t2;
```

```
    }
       }
    id<<=1;
       i1=id-n2;
       id<<=1;
    } while ( i1<n );
  a=e;
  for (j=2; j<=n8; j++){
        a3=3*a;
        cc1=cos(a);
        ss1=sin(a);
        cc3=cos(a3);
        ss3=sin(a3);
        a=j*e;
        i=0;
        id=n2<<1;
        do{
      for (; i<n; i+=id){
      i1=i+j-1;
      i2=i1+n4;
      i3=i2+n4;
      i4=i3+n4;
      i5=i+n4-j+1;
      i6=i5+n4;
      i7=i6+n4;
      i8=i7+n4;
      t1=data[i3]*cc1+data[i7]*ss1;
      t2=data[i7]*cc1-data[i3]*ss1;
      t3=data[i4]*cc3+data[i8]*ss3;
      t4=data[i8]*cc3-data[i4]*ss3;
      t5=t1+t3;
      t6=t2+t4;
      t3=t1-t3;
      t4=t2-t4;
      t2=data[i6]+t6;
      data[i3]=t6-data[i6];
      data[i8]=t2;
      t2=data[i2]-t3;
      data[i7]=-data[i2]-t3;
      data[i4]=t2;
      t1=data[i1]+t5;
      data[i6]=data[i1]-t5;
      data[i1]=t1;
      t1=data[i5]+t4;
      data[i5]-=t4;
      data[i2]=t1;
        }
      id<<=1;
      i=id-n2;
      id<<=1;
    } while(i<n);
     }
       }

  //division with array length
    for(i=0;i<n;i++) data[i]/=n;
}


/////////////////////////////////////////////////////////
// Sorensen in-place split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source:
// Source: see the routines it calls ...

void realfft_split_unshuffled(double *data,long n){

 double *data2;
 long i,j;

 realfft_split(data,n);
 //unshuffling - not in-place
 data2=(double *)malloc(n*sizeof(double));
 j=n/2;
 data2[0]=data[0];
 data2[1]=data[j];
 for(i=1;i<j;i++) {data2[i+i]=data[i];data2[i+i+1]=data[n-i];}
```

```
 for(i=0;i<n;i++) data[i]=data2[i];
 free(data2);
}

/////////////////////////////////////////////////////////
// Sorensen in-place inverse split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size/2),im(size/2-1),...,im(1)
//
// output:
// re(0),re(1),re(2),...,re(size-1)
// NOT normalized by array length
//
// Source:
// Sorensen et al: Real-Valued Fast Fourier Transform Algorithms,
// IEEE Trans. ASSP, ASSP-35, No. 6, June 1987

void irealfft_split(double *data,long n){

  long i,j,k,i5,i6,i7,i8,i0,id,i1,i2,i3,i4,n2,n4,n8,n1;
  double t1,t2,t3,t4,t5,a3,ss1,ss3,cc1,cc3,a,e,sqrt2;

  sqrt2=sqrt(2.0);

n1=n-1;
n2=n<<1;
for(k=n;k>2;k>>=1){
 id=n2;
 n2>>=1;
 n4=n2>>2;
 n8=n2>>3;
 e = 2*pi/(n2);
 i1=0;
 do{
     for (; i1<n; i1+=id){
   i2=i1+n4;
   i3=i2+n4;
   i4=i3+n4;
   t1=data[i1]-data[i3];
   data[i1]+=data[i3];
   data[i2]*=2;
   data[i3]=t1-2*data[i4];
   data[i4]=t1+2*data[i4];
   if (n4!=1){
    i0=i1+n8;
    i2+=n8;
    i3+=n8;
    i4+=n8;
    t1=(data[i2]-data[i0])/sqrt2;
    t2=(data[i4]+data[i3])/sqrt2;
    data[i0]+=data[i2];
    data[i2]=data[i4]-data[i3];
    data[i3]=2*(-t2-t1);
    data[i4]=2*(-t2+t1);
   }
      }
   id<<=1;
      i1=id-n2;
      id<<=1;
   } while ( i1<n1 );
 a=e;
 for (j=2; j<=n8; j++){
       a3=3*a;
       cc1=cos(a);
       ss1=sin(a);
       cc3=cos(a3);
       ss3=sin(a3);
       a=j*e;
       i=0;
       id=n2<<1;
       do{
     for (; i<n; i+=id){
     i1=i+j-1;
     i2=i1+n4;
     i3=i2+n4;
     i4=i3+n4;
     i5=i+n4-j+1;
     i6=i5+n4;
     i7=i6+n4;
     i8=i7+n4;
     t1=data[i1]-data[i6];
     data[i1]+=data[i6];
     t2=data[i5]-data[i2];
```

```
      data[i5]+=data[i2];
      t3=data[i8]+data[i3];
      data[i6]=data[i8]-data[i3];
      t4=data[i4]+data[i7];
      data[i2]=data[i4]-data[i7];
      t5=t1-t4;
      t1+=t4;
      t4=t2-t3;
      t2+=t3;
      data[i3]=t5*cc1+t4*ss1;
      data[i7]=-t4*cc1+t5*ss1;
      data[i4]=t1*cc3-t2*ss3;
      data[i8]=t2*cc3+t1*ss3;
      }
      id<<=1;
      i=id-n2;
      id<<=1;
    } while(i<n1);
     }
 }

    /*----------------------*/
 i0=0;
 id=4;
    do{
        for (; i0<n1; i0+=id){
    i1=i0+1;
    t1=data[i0];
    data[i0]=t1+data[i1];
    data[i1]=t1-data[i1];
   }
     id<<=1;
        i0=id-2;
        id<<=1;
     } while ( i0<n1 );

/*----------------------*/

//data shuffling
      for (i=0,j=0,n2=n/2; i<n1 ; i++){
    if (i<j){
     t1=data[j];
     data[j]=data[i];
     data[i]=t1;
     }
    k=n2;
    while (k<=j){
     j-=k;
     k>>=1;
     }
    j+=k;
      }
}


//////////////////////////////////////////////////////////
// Sorensen in-place radix-2 FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(1),re(2),...,re(size/2),im(size/2-1),...,im(1)
// normalized by array length
//
// Source:
// Sorensen et al: Real-Valued Fast Fourier Transform Algorithms,
// IEEE Trans. ASSP, ASSP-35, No. 6, June 1987

void realfft_radix2(double *data,long n){

    double  xt,a,e, t1, t2, cc, ss;
    long  i, j, k, n1, n2, n3, n4, i1, i2, i3, i4;

 n4=n-1;
  //data shuffling
      for (i=0,j=0,n2=n/2; i<n4 ; i++){
    if (i<j){
     xt=data[j];
     data[j]=data[i];
     data[i]=xt;
     }
    k=n2;
    while (k<=j){
```

```
    j-=k;
    k>>=1;
    }
   j+=k;
      }

/* -------------------- */
    for (i=0; i<n; i += 2)
      {
  xt = data[i];
  data[i] = xt + data[i+1];
  data[i+1] = xt - data[i+1];
      }
/* ------------------------ */
    n2 = 1;
    for (k=n;k>2;k>>=1){
  n4 = n2;
  n2 = n4 << 1;
  n1 = n2 << 1;
  e = 2*pi/(n1);
  for (i=0; i<n; i+=n1){
   xt = data[i];
   data[i] = xt + data[i+n2];
   data[i+n2] = xt-data[i+n2];
   data[i+n4+n2] = -data[i+n4+n2];
   a = e;
   n3=n4-1;
   for (j = 1; j <=n3; j++){
    i1 = i+j;
    i2 = i - j + n2;
    i3 = i1 + n2;
    i4 = i - j + n1;
    cc = cos(a);
    ss = sin(a);
    a += e;
    t1 = data[i3] * cc + data[i4] * ss;
    t2 = data[i3] * ss - data[i4] * cc;
    data[i4] = data[i2] - t2;
    data[i3] = -data[i2] - t2;
    data[i2] = data[i1] - t1;
    data[i1] += t1;
    }
   }
  }

 //division with array length
   for(i=0;i<n;i++) data[i]/=n;
}


/////////////////////////////////////////////////////////
// Sorensen in-place split-radix FFT for real values
// data: array of doubles:
// re(0),re(1),re(2),...,re(size-1)
//
// output:
// re(0),re(size/2),re(1),im(1),re(2),im(2),...,re(size/2-1),im(size/2-1)
// normalized by array length
//
// Source:
// Source: see the routines it calls ...

void realfft_radix2_unshuffled(double *data,long n){

 double *data2;
 long i,j;

 //unshuffling - not in-place
 realfft_radix2(data,n);
 data2=(double *)malloc(n*sizeof(double));
 j=n/2;
 data2[0]=data[0];
 data2[1]=data[j];
 for(i=1;i<j;i++) {data2[i+i]=data[i];data2[i+i+1]=data[n-i];}
 for(i=0;i<n;i++) data[i]=data2[i];
 free(data2);
}
```

FFT classes in C++ and Object Pascal (click this to go back to the index)

Type : Real-to-Complex FFT and Complex-to-Real IFFT

References : Laurent de Soras (Object Pascal translation by Frederic Vanmol)

Linked file : FFTReal.zip

Notes :
(see linkfile)

Comments
from : ms_shirbiny@hotmail.com
comment : the file doesn't exist

Java FFT (click this to go back to the index)

Type : FFT Analysis

References : Posted by Loreno Heer

Notes :
May not work correctly ;-)

Code :
```
// WTest.java
/*
    Copyright (C) 2003 Loreno Heer, (helohe at bluewin dot ch)

    This program is free software; you can redistribute it and/or modify
 it under the terms of the GNU General Public License as published by
 the Free Software Foundation; either version 2 of the License, or
 (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

*/
public class WTest{

 private static double[] sin(double step, int size){
  double f = 0;
  double[] ret = new double[size];
  for(int i = 0; i < size; i++){
   ret[i] = Math.sin(f);
   f += step;
  }
  return ret;
 }

 private static double[] add(double[] a, double[] b){
  double[] c = new double[a.length];
  for(int i = 0; i < a.length; i++){
   c[i] = a[i] + b[i];
  }
  return c;
 }

 private static double[] sub(double[] a, double[] b){
  double[] c = new double[a.length];
  for(int i = 0; i < a.length; i++){
   c[i] = a[i] - b[i];
  }
  return c;
 }

 private static double[] add(double[] a, double b){
  double[] c = new double[a.length];
  for(int i = 0; i < a.length; i++){
   c[i] = a[i] + b;
  }
  return c;
 }

 private static double[] cp(double[] a, int size){
  double[] c = new double[size];
  for(int i = 0; i < size; i++){
   c[i] = a[i];
  }
  return c;
 }

 private static double[] mul(double[] a, double b){
  double[] c = new double[a.length];
  for(int i = 0; i < a.length; i++){
   c[i] = a[i] * b;
  }
  return c;
 }

 private static void print(double[] value){
  for(int i = 0; i < value.length; i++){
   System.out.print(i + "," + value[i] + "\n");
  }
  System.out.println();
 }

 private static double abs(double[] a){
  double c = 0;
  for(int i = 0; i < a.length; i++){
```

```
   c = ((c * i) + Math.abs(a[i])) / (i + 1);
  }
  return c;
 }

 private static double[] fft(double[] a, int min, int max, int step){
  double[] ret = new double[(max - min) / step];
  int i = 0;
  for(int d = min; d < max; d = d + step){
   double[] f = sin(fc(d), a.length);
   double[] dif = sub(a, f);
   ret[i] = 1 - abs(dif);
   i++;
  }
  return ret;
 }

 private static double[] fft_log(double[] a){
  double[] ret = new double[1551];
  int i = 0;
  for(double d = 0; d < 15.5; d = d + 0.01){
   double[] f = sin(fc(Math.pow(2,d)), a.length);
   double[] dif = sub(a, f);
   ret[i] = Math.abs(1 - abs(dif));
   i++;
  }
  return ret;
 }

 private static double fc(double d){
  return d * Math.PI / res;
 }

 private static void print_log(double[] value){
  for(int i = 0; i < value.length; i++){
   System.out.print(Math.pow(2,((double)i/100d)) + "," + value[i] + "\n");
  }
  System.out.println();
 }

 public static void main(String[] args){
  double[] f_0 = sin(fc(440), sample_length); // res / pi =>14005
  //double[] f_1 = sin(.02, sample_length);
  double[] f_2 = sin(fc(520), sample_length);
  //double[] f_3 = sin(.25, sample_length);

  //double[] f = add( add( add(f_0, f_1), f_2), f_3);

  double[] f = add(f_0, f_2);

  //print(f);

  double[] d = cp(f,1000);
  print_log(fft_log(d));
 }

 static double length = .2; // sec
 static int res = 44000; // resoultion (pro sec)
 static int sample_length = res; // resoultion

}
```

Look ahead limiting (click this to go back to the index)

References : Posted by Wilfried Welti

Notes :
use add_value with all values which enter the look-ahead area,
and remove_value with all value which leave this area. to get
the maximum value in the look-ahead area, use get_max_value.
in the very beginning initialize the table with zeroes.

If you always want to know the maximum amplitude in
your look-ahead area, the thing becomes a sorting
problem. very primitive approach using a look-up table

Code :
```
void lookup_add(unsigned section, unsigned size, unsigned value)
{
  if (section==value)
    lookup[section]++;
  else
  {
    size >>= 1;
    if (value>section)
    {
      lookup[section]++;
      lookup_add(section+size,size,value);
    }
    else
      lookup_add(section-size,size,value);
  }
}

void lookup_remove(unsigned section, unsigned size, unsigned value)
{
  if (section==value)
    lookup[section]--;
  else
  {
    size >>= 1;
    if (value>section)
    {
      lookup[section]--;
      lookup_remove(section+size,size,value);
    }
    else
      lookup_remove(section-size,size,value);
  }
}

unsigned lookup_getmax(unsigned section, unsigned size)
{
  unsigned max = lookup[section] ? section : 0;
  size >>= 1;
  if (size)
    if (max)
    {
      max = lookup_getmax((section+size),size);
      if (!max) max=section;
    }
    else
      max = lookup_getmax((section-size),size);
  return max;
}

void add_value(unsigned value)
{
  lookup_add(LOOKUP_VALUES>>1, LOOKUP_VALUES>>1, value);
}

void remove_value(unsigned value)
{
  lookup_remove(LOOKUP_VALUES>>1, LOOKUP_VALUES>>1, value);
}

unsigned get_max_value()
{
  return lookup_getmax(LOOKUP_VALUES>>1, LOOKUP_VALUES>>1);
}
```

LPC analysis (autocorrelation + Levinson-Durbin recursion) (click this to go back to the index)

References : Posted by mail@mutagene.net

Notes :
The autocorrelation function implements a warped autocorrelation, so that frequency resolution can be specified by the variable 'lambda'. Levinson-Durbin recursion calculates autoregression coefficients a and reflection coefficients (for lattice filter implementation) K. Comments for Levinson-Durbin function implement matlab version of the same function.

No optimizations.

Code :
```
//find the order-P autocorrelation array, R, for the sequence x of length L and warping of lambda
//wAutocorrelate(&pfSrc[stIndex],siglen,R,P,0);
wAutocorrelate(float * x, unsigned int L, float * R, unsigned int P, float lambda)
{
 double * dl = new double [L];
 double * Rt = new double [L];
 double r1,r2,r1t;
 R[0]=0;
 Rt[0]=0;
 r1=0;
 r2=0;
 r1t=0;
 for(unsigned int k=0; k<L;k++)
 {
   Rt[0]+=double(x[k])*double(x[k]);

   dl[k]=r1-double(lambda)*double(x[k]-r2);
   r1 = x[k];
   r2 = dl[k];
 }
 for(unsigned int i=1; i<=P; i++)
 {
  Rt[i]=0;
  r1=0;
  r2=0;
  for(unsigned int k=0; k<L;k++)
  {
   Rt[i]+=double(dl[k])*double(x[k]);

   r1t = dl[k];
   dl[k]=r1-double(lambda)*double(r1t-r2);
   r1 = r1t;
   r2 = dl[k];
  }
 }
 for(i=0; i<=P; i++)
  R[i]=float(Rt[i]);
 delete[] dl;
 delete[] Rt;
}

// Calculate the Levinson-Durbin recursion for the autocorrelation sequence R of length P+1 and return the
autocorrelation coefficients a and reflection coefficients K
LevinsonRecursion(unsigned int P, float *R, float *A, float *K)
{
 double Am1[62];

 if(R[0]==0.0) {
  for(unsigned int i=1; i<=P; i++)
  {
   K[i]=0.0;
   A[i]=0.0;
  }}
    else {
  double km,Em1,Em;
  unsigned int k,s,m;
  for (k=0;k<=P;k++){
   A[0]=0;
   Am1[0]=0; }
  A[0]=1;
  Am1[0]=1;
  km=0;
  Em1=R[0];
  for (m=1;m<=P;m++)     //m=2:N+1
  {
   double err=0.0f;     //err = 0;
   for (k=1;k<=m-1;k++)   //for k=2:m-1
    err += Am1[k]*R[m-k];  // err = err + am1(k)*R(m-k+1);
   km = (R[m]-err)/Em1;   //km=(R(m)-err)/Em1;
   K[m-1] = -float(km);
   A[m]=(float)km;      //am(m)=km;
   for (k=1;k<=m-1;k++)   //for k=2:m-1
    A[k]=float(Am1[k]-km*Am1[m-k]);  // am(k)=am1(k)-km*am1(m-k+1);
   Em=(1-km*km)*Em1;    //Em=(1-km*km)*Em1;
   for(s=0;s<=P;s++)    //for s=1:N+1
    Am1[s] = A[s];       // am1(s) = am(s)
   Em1 = Em;      //Em1 = Em;
  }
 }
```

```
 return 0;
}
```

Magnitude and phase plot of arbitrary IIR function, up to 5th order (click this to go back to the index)

Type : magnitude and phase at any frequency

References : Posted by George Yohng

Notes :
Amplitude and phase calculation of IIR equation
run at sample rate "sampleRate" at frequency "F".

AMPLITUDE
----
cf_mag(F,sampleRate,
  a0,a1,a2,a3,a4,a5,
  b0,b1,b2,b3,b4,b5)

PHASE
----
cf_phi(F,sampleRate,
  a0,a1,a2,a3,a4,a5,
  b0,b1,b2,b3,b4,b5)

If you need a frequency diagram, draw a plot for
F=0...sampleRate/2

If you need amplitude in dB, use cf_lin2db(cf_mag(.......))

Set b0=-1 if you have such function:

y[n] = a0*x[n] + a1*x[n-1] + a2*x[n-2] + a3*x[n-3] + a4*x[n-4] + a5*x[n-5] +
  + b1*y[n-1] + b2*y[n-2] + b3*y[n-3] + b4*y[n-4] + b5*y[n-5];

Set b0=1 if you have such function:

y[n] = a0*x[n] + a1*x[n-1] + a2*x[n-2] + a3*x[n-3] + a4*x[n-4] + a5*x[n-5] +
  - b1*y[n-1] - b2*y[n-2] - b3*y[n-3] - b4*y[n-4] - b5*y[n-5];

Do not try to reverse engineer these formulae - they don't give any sense
other than they are derived from transfer function, and they work. :)

Code :
```
/*
 C file can be downloaded from
 http://www.yohng.com/dsp/cfsmp.c
*/


#define C_PI 3.14159265358979323846264

double cf_mag(double f,double rate,
              double a0,double a1,double a2,double a3,double a4,double a5,
              double b0,double b1,double b2,double b3,double b4,double b5)
{
    return
        sqrt((a0*a0 + a1*a1 + a2*a2 + a3*a3 + a4*a4 + a5*a5 +
              2*(a0*a1 + a1*a2 + a2*a3 + a3*a4 + a4*a5)*cos((2*f*C_PI)/rate) +
              2*(a0*a2 + a1*a3 + a2*a4 + a3*a5)*cos((4*f*C_PI)/rate) +
              2*a0*a3*cos((6*f*C_PI)/rate) + 2*a1*a4*cos((6*f*C_PI)/rate) +
              2*a2*a5*cos((6*f*C_PI)/rate) + 2*a0*a4*cos((8*f*C_PI)/rate) +
              2*a1*a5*cos((8*f*C_PI)/rate) + 2*a0*a5*cos((10*f*C_PI)/rate))/
              (b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +
              2*(b0*b1 + b1*b2 + b2*b3 + b3*b4 + b4*b5)*cos((2*f*C_PI)/rate) +
              2*(b0*b2 + b1*b3 + b2*b4 + b3*b5)*cos((4*f*C_PI)/rate) +
              2*b0*b3*cos((6*f*C_PI)/rate) + 2*b1*b4*cos((6*f*C_PI)/rate) +
              2*b2*b5*cos((6*f*C_PI)/rate) + 2*b0*b4*cos((8*f*C_PI)/rate) +
              2*b1*b5*cos((8*f*C_PI)/rate) + 2*b0*b5*cos((10*f*C_PI)/rate)));
}


double cf_phi(double f,double rate,
              double a0,double a1,double a2,double a3,double a4,double a5,
              double b0,double b1,double b2,double b3,double b4,double b5)
{
        atan2((a0*b0 + a1*b1 + a2*b2 + a3*b3 + a4*b4 + a5*b5 +
              (a0*b1 + a1*(b0 + b2) + a2*(b1 + b3) + a5*b4 + a3*(b2 + b4) +
              a4*(b3 + b5))*cos((2*f*C_PI)/rate) +
              ((a0 + a4)*b2 + (a1 + a5)*b3 + a2*(b0 + b4) +
              a3*(b1 + b5))*cos((4*f*C_PI)/rate) + a3*b0*cos((6*f*C_PI)/rate) +
              a4*b1*cos((6*f*C_PI)/rate) + a5*b2*cos((6*f*C_PI)/rate) +
              a0*b3*cos((6*f*C_PI)/rate) + a1*b4*cos((6*f*C_PI)/rate) +
              a2*b5*cos((6*f*C_PI)/rate) + a4*b0*cos((8*f*C_PI)/rate) +
              a5*b1*cos((8*f*C_PI)/rate) + a0*b4*cos((8*f*C_PI)/rate) +
              a1*b5*cos((8*f*C_PI)/rate) +
              (a5*b0 + a0*b5)*cos((10*f*C_PI)/rate))/
```

```
              (b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +
              2*((b0*b1 + b1*b2 + b3*(b2 + b4) + b4*b5)*cos((2*f*C_PI)/rate) +
              (b2*(b0 + b4) + b3*(b1 + b5))*cos((4*f*C_PI)/rate) +
              (b0*b3 + b1*b4 + b2*b5)*cos((6*f*C_PI)/rate) +
              (b0*b4 + b1*b5)*cos((8*f*C_PI)/rate) +
              b0*b5*cos((10*f*C_PI)/rate))),

             ((a1*b0 + a3*b0 + a5*b0 - a0*b1 + a2*b1 + a4*b1 - a1*b2 +
              a3*b2 + a5*b2 - a0*b3 - a2*b3 + a4*b3 -
              a1*b4 - a3*b4 + a5*b4 - a0*b5 - a2*b5 - a4*b5 +
              2*(a3*b1 + a5*b1 - a0*b2 + a4*(b0 + b2) - a1*b3 + a5*b3 +
              a2*(b0 - b4) - a0*b4 - a1*b5 - a3*b5)*cos((2*f*C_PI)/rate) +
              2*(a3*b0 + a4*b1 + a5*(b0 + b2) - a0*b3 - a1*b4 - a0*b5 - a2*b5)*
              cos((4*f*C_PI)/rate) + 2*a4*b0*cos((6*f*C_PI)/rate) +
              2*a5*b1*cos((6*f*C_PI)/rate) - 2*a0*b4*cos((6*f*C_PI)/rate) -
              2*a1*b5*cos((6*f*C_PI)/rate) + 2*a5*b0*cos((8*f*C_PI)/rate) -
              2*a0*b5*cos((8*f*C_PI)/rate))*sin((2*f*C_PI)/rate))/
              (b0*b0 + b1*b1 + b2*b2 + b3*b3 + b4*b4 + b5*b5 +
              2*(b0*b1 + b1*b2 + b2*b3 + b3*b4 + b4*b5)*cos((2*f*C_PI)/rate) +
              2*(b0*b2 + b1*b3 + b2*b4 + b3*b5)*cos((4*f*C_PI)/rate) +
              2*b0*b3*cos((6*f*C_PI)/rate) + 2*b1*b4*cos((6*f*C_PI)/rate) +
              2*b2*b5*cos((6*f*C_PI)/rate) + 2*b0*b4*cos((8*f*C_PI)/rate) +
              2*b1*b5*cos((8*f*C_PI)/rate) + 2*b0*b5*cos((10*f*C_PI)/rate)));
}

double cf_lin2db(double lin)
{
    if (lin<9e-51) return -1000; /* prevent invalid operation */
    return 20*log10(lin);
}
```

Comments
from :
comment : They don't appear to make any sense at all.

from : Rob
comment : Actually it is simpler to simply take the zero-padded b and a coefficients and do real->complex FFT like this (matlab code):

H_complex=fft(b,N)./fft(a,N);
phase=angle(H_complex);
Magn=abs(H_complex);

This will give you N/2 points from 0 to pi angle freq (or 0 to nyquist freq).

/Rob

Measuring interpollation noise (click this to go back to the index)

References : Posted by Jon Watte

Notes :
You can easily estimate the error by evaluating the actual function and evaluating your interpolator at each of the mid-points between your samples. The absolute difference between these values, over the absolute value of the "correct" value, is your relative error. log10 of your relative error times 20 is an estimate of your quantization noise in dB. Example:

You have a table for every 0.5 "index units". The value at index unit 72.0 is 0.995 and the value at index unit 72.5 is 0.999. The interpolated value at index 72.25 is 0.997. Suppose the actual function value at that point was 0.998; you would have an error of 0.001 which is a relative error of 0.001002004.. log10(error) is about -2.99913, which times 20 is about -59.98. Thus, that's your quantization noise at that position in the table. Repeat for each pair of samples in the table.

Note: I said "quantization noise" not "aliasing noise". The aliasing noise will, as far as I know, only happen when you start up-sampling without band-limiting and get frequency aliasing (wrap-around), and thus is mostly independent of what specific interpolation mechanism you're using.

QFT and DQFT (double precision) classes (click this to go back to the index)

References : Posted by Joshua Scholar

Linked file : qft.tar_1.gz

Notes :
Since it's a Visual C++ project (though it has relatively portable C++) I
guess the main audience are PC users. As such I'm including a zip file.
Some PC users wouldn't know what to do with a tgz file.

The QFT and DQFT (double precision) classes supply the following functions:

1. Real valued FFT and inverse FFT functions. Note that separate arraysare used
   for real and imaginary component of the resulting spectrum.

2. Decomposition of a spectrum into a separate spectrum of the evensamples
   and a spectrum of the odd samples. This can be useful for buildingfilter banks.

3. Reconstituting a spectrum from separate spectrums of the even samples
   and odd samples. This can be useful for building filter banks.

4. A discrete Sin transform (a QFT decomposes an FFT into a DST and DCT).

5. A discrete Cos transfrom.

6. Since a QFT does it's last stage calculating from the outside in thelast part
   can be left unpacked and only calculated as needed in the case wherethe entire
spectrum isn't needed (I used this for calculating correlations andconvolutions
where I only needed half of the results).
ReverseNoUnpack()
UnpackStep()
and NegUnpackStep()
implement this functionality

NOTE Reverse() normalizes its results (divides by one half the blocklength), but
ReverseNoUnpack() does not.

7. Also if you only want the first half of the results you can call ReverseHalf()

NOTE Reverse() normalizes its results (divides by one half the blocklength), but
ReverseHalf() does not.

8. QFT is less numerically stable than regular FFTs. With singleprecision calculations,
   a block length of 2^15 brings the accuracy down to being barelyaccurate enough.
At that size, single precision calculations tested sound files wouldoccasionally have
a sample off by 2, and a couple off by 1 per block. Full volume whitenoise would generate
a few samples off by as much as 6 per block at the end, beginning and middle.

No matter what the inputs the errors are always at the same positions in the block.
There some sort of cancelation that gets more delicate as the block size gets bigger.

For the sake of doing convolutions and the like where the forward transform is
done only once for one of the inputs, I created a AccurateForward() function.
It uses a regular FFT algorithm for blocks larger than 2^12, and decomposes into even and
odd FFTs recursively.

In any case you can always use the double precision routines to get more accuracy.
DQFT even has routines that take floats as inputs and return double precision
spectrum outputs.

As for portability:

1. The files qft.cpp and dqft.cpp start with defines:
#define _USE_ASM

If you comment those define out, then what's left is C++ with no assembly language.

2. There is unnecessary windows specific code in "criticalSection.h"
I used a critical section because objects are not reentrant (each object has
permanent scratch pad memory), but obviously critical sections are operating
system specific. In any case that code can easily be taken out.

If you look at my code and see that there's an a test built in the examples
that makes sure that the results are in the ballpark of being right. It
wasn't that I expected the answers to be far off, it was that I uncommenting
the "no assembly language" versions of some routines and I wanted to make
sure that they weren't broken.

Simple peak follower (click this to go back to the index)

Type : amplitude analysis

References : Posted by Phil Burk

Notes :
This simple peak follower will give track the peaks of a signal. It will rise rapidly when the input is rising, and then decay exponentially when the input drops. It can be used to drive VU meters, or used in an automatic gain control circuit.

Code :
```
// halfLife = time in seconds for output to decay to half value after an impulse

static float output = 0.0;

float scalar = pow( 0.5, 1.0/(halfLife * sampleRate)));

if( input < 0.0 )
  input = -input;  /* Absolute value. */

if ( input >= output )
{
   /* When we hit a peak, ride the peak to the top. */
   output = input;
}
else
{
   /* Exponential decay of output when signal is low. */
   output = output * scalar;
   /*
   ** When current gets close to 0.0, set current to 0.0 to prevent FP underflow
   ** which can cause a severe performance degradation due to a flood
   ** of interrupts.
   */
   if( output < VERY_SMALL_FLOAT ) output = 0.0;
}
```

tone detection with Goertzel (click this to go back to the index)

Type : Goertzel

References : Posted by espenr@ii.uib.no

Linked file : http://www.ii.uib.no/~espenr/tonedetect.zip

Notes :
Goertzel is basically DFT of parts of a spectrum not the total spectrum as you normally do with FFT. So if you just want to check out the power for some frequencies this could be better. Is good for DTFM detection I've heard.

The WNk isn't calculated 100% correctly, but it seems to work so ;) Yeah and the code is C++ so you might have to do some small adjustment to compile it as C.

Code :
```
/** Tone detect by Goertzel algorithm
 *
 * This program basically searches for tones (sines) in a sample and reports the different dB it finds for
 * different frequencies. Can easily be extended with some thresholding to report true/false on detection.
 * I'm far from certain goertzel it implemented 100% correct, but it works :)
 *
 * Hint, the SAMPLERATE, BUFFERSIZE, FREQUENCY, NOISE and SIGNALVOLUME all affects the outcome of the reported
dB. Tweak
 * em to find the settings best for your application. Also, seems to be pretty sensitive to noise (whitenoise
anyway) which
 * is a bit sad. Also I don't know if the goertzel really likes float values for the frequency ... And using
44100 as
 * samplerate for detecting 6000 Hz tone is kinda silly I know :)
 *
 * Written by: Espen Riskedal, espenr@ii.uib.no, july-2002
 */

#include <iostream>
#include <cmath>
#include <cstdlib>

using std::rand;
// math stuff
using std::cos;
using std::abs;
using std::exp;
using std::log10;
// iostream stuff
using std::cout;
using std::endl;

#define PI 3.14159265358979323844
// change the defines if you want to
#define SAMPLERATE 44100
#define BUFFERSIZE 8820
#define FREQUENCY 6000
#define NOISE 0.05
#define SIGNALVOLUME 0.8

/**  The Goertzel algorithm computes the k-th DFT coefficient of the input signal using a second-order filter.
 *   http://ptolemy.eecs.berkeley.edu/papers/96/dtmf_ict/www/node3.html.
 *   Basiclly it just does a DFT of the frequency we want to check, and none of the others (FFT calculates for
all frequencies).
 */
float goertzel(float *x, int N, float frequency, int samplerate) {
    float Skn, Skn1, Skn2;
    Skn = Skn1 = Skn2 = 0;

    for (int i=0; i<N; i++) {
 Skn2 = Skn1;
 Skn1 = Skn;
 Skn = 2*cos(2*PI*frequency/samplerate)*Skn1 - Skn2 + x[i];
    }

    float WNk = exp(-2*PI*frequency/samplerate); // this one ignores complex stuff
    //float WNk = exp(-2*j*PI*k/N);
    return (Skn - WNk*Skn1);
}

/** Generates a tone of the specified frequency
 *  Gotten from: http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&oe=UTF-
8&safe=off&selm=3c641e%243jn%40uicsl.csl.uiuc.edu
 */
float *makeTone(int samplerate, float frequency, int length, float gain=1.0) {
    //y(n) = 2 * cos(A) * y(n-1) - y(n-2)
    //A= (frequency of interest) * 2 * PI / (sampling frequency)
    //A is in radians.
    // frequency of interest MUST be <= 1/2 the sampling frequency.
    float *tone = new float[length];
    float A = frequency*2*PI/samplerate;

    for (int i=0; i<length; i++) {
 if (i > 1) tone[i]= 2*cos(A)*tone[i-1] - tone[i-2];
 else if (i > 0) tone[i] = 2*cos(A)*tone[i-1] - (cos(A));
```

```
 else tone[i] = 2*cos(A)*cos(A) - cos(2*A);
    }

    for (int i=0; i<length; i++) tone[i] = tone[i]*gain;

    return tone;
}

/** adds whitenoise to a sample */
void *addNoise(float *sample, int length, float gain=1.0) {
    for (int i=0; i<length; i++) sample[i] += (2*(rand()/(float)RAND_MAX)-1)*gain;
}

/** returns the signal power/dB */
float power(float value) {
    return 20*log10(abs(value));
}

int main(int argc, const char* argv) {
    cout << "Samplerate: " << SAMPLERATE << "Hz\n";
    cout << "Buffersize: " << BUFFERSIZE << " samples\n";
    cout << "Correct frequency is: " << FREQUENCY << "Hz\n";
    cout << " - signal volume: " << SIGNALVOLUME*100 << "%\n";
    cout << " - white noise: " << NOISE*100 << "%\n";

    float *tone = makeTone(SAMPLERATE, FREQUENCY, BUFFERSIZE, SIGNALVOLUME);
    addNoise(tone, BUFFERSIZE,NOISE);

    int stepsize = FREQUENCY/5;

    for (int i=0; i<10; i++) {
 int freq = stepsize*i;
 cout << "Trying freq: " << freq << "Hz  ->  dB: " << power(goertzel(tone, BUFFERSIZE, freq, SAMPLERATE)) <<
endl;
    }
    delete tone;

    return 0;
}
```

Comments

from : ashg@eth.net
comment : Hello!
I am interested in knowing that could we implement the Goertzel algorithm using only integer variables and not using floa at all. Please let me know. I need it urgently.
regards,
Ashish

from : yunusk@telesis.com.tr
comment : Hello,

I will implement DTMF/MFR1/MFR2 generation/detection function using DSP. I have found goertzel algorithm for DTMF. Can I use this algorithm for MFR1 and MFR2? Could you please let me know ?
Best Regards
Yunus

from : asaeiaf@hotmail.com
comment : Hello
I'm going to implement a Goertzel algorithm on a Fixed_point DSP .
Could you lead me to the changes I should consider?
I realy appreciate your helping me.

regards,
Afsaneh Asaei

from : jfishman@umsis.miamil.edu
comment : Does anybody know if and how this can be done in real time?

Thanks,
JF

from : no
comment : It can.

from : pabitra_mohan208@yahoomail.com
comment : sir.
i m pabitra.
please help me to understand this program.

18dB/oct resonant 3 pole LPF with tanh() dist (click this to go back to the index)

References : Posted by Josep M Comajuncosas

Linked file : lpf18.zip

Linked file : lpf18.sme

Notes :
Implementation in CSound and Sync Modular...

1st and 2nd order pink noise filters (click this to go back to the index)

Type : Pink noise

References : Posted by umminger@umminger.com

Notes :
Here are some new lower-order pink noise filter coefficients.

These have approximately equiripple error in decibels from 20hz to 20khz at a 44.1khz sampling rate.

1st order, ~ +/- 3 dB error (not recommended!)
num = [0.05338071119116 -0.03752455712906]
den = [1.00000000000000 -0.97712493947102]

2nd order, ~ +/- 0.9 dB error
num = [0.04957526213389 -0.06305581334498   0.01483220320740]
den = [1.00000000000000 -1.80116083982126   0.80257737639225]

303 type filter with saturation (click this to go back to the index)

Type : Runge-Kutta Filters

References : Posted by Hans Mikelson

Linked file : filters001.txt (this linked file is included below)

Notes :
I posted a filter to the Csound mailing list a couple of weeks ago that has a 303 flavor to it. It basically does wacky distortions to the sound. I used Runge-Kutta for the diff eq. simulation though which makes it somewhat sluggish.

This is a CSound score!!

Linked files

```
; ORCHESTRA
;---------------------------------------------------------
; Runge-Kutta Filters
; Coded by Hans Mikelson June, 2000
;---------------------------------------------------------
sr       =        44100                       ; Sample rate
kr       =        44100                       ; Kontrol rate
ksmps    =        1                           ; Samples/Kontrol period
nchnls   =        2                           ; Normal stereo
         zakinit  50, 50

;------------------------------------------------------------
; Envelope (Knob twisting simulation)
;------------------------------------------------------------
       instr  1

idur   =      p3          ; Duration
iamp   =      p4          ; Amplitude
ilps   =      p5          ; Loops
iofst  =      p6          ; Offset
itabl  =      p7          ; Table
ioutch =      p8          ; Output channel
iphase =      p9          ; Phase

kout   oscili iamp, ilps/idur, itabl, iphase  ; Create the envelope
       zkw    kout+iofst, ioutch      ; Send out to the zak channel

      endin

;---------------------------------------------------------
; Runge-Kutta Freaky Filter
;---------------------------------------------------------
       instr  7

idur   =      p3             ; Duration
iamp   =      p4             ; Amplitude
kfco2  zkr    p5             ; Filter cutoff
kq1    zkr    p6             ; Q
ih     =      .001           ; Diff eq step size
ipanl  =      sqrt(p8)       ; Pan left
ipanr  =      sqrt(1-p8)     ; Pan right
ifqc   =      cpspch(p9)     ; Pitch to frequency
kpb    zkr    p10            ; Pentic bounce frequency
kpa    zkr    p11            ; Pentic bounce amount
kasym  zkr    p12            ; Q assymmetry amount
kasep  zkr    p13            ; Q asymmetry separation


kdclck linseg 0, .02, 1, idur-.04, 1, .02, 0 ; Declick envelope
kfco1  expseg 1, idur, .1
kfco   =      kfco1*kfco2
kq     =      kq1*kfco^1.2*.1
kfc    =      kfco/8/sr*44100

ay     init   0
ay1    init   0
ay2    init   0
ay3    init   0
axs    init   0
avxs   init   0
ax     vco   1, ifqc, 2, 1, 1, 1        ; Square wave

  ; R-K Section 1
  afdbk =      kq*ay/(1+exp(-ay*3*kasep)*kasym)   ; Only oscillate in one
direction
```

```
  ak11  =      ih*((ax-ay1)*kfc-afdbk)
  ak21  =      ih*((ax-(ay1+.5*ak11))*kfc-afdbk)
  ak31  =      ih*((ax-(ay1+.5*ak21))*kfc-afdbk)
  ak41  =      ih*((ax-(ay1+ak31))*kfc-afdbk)
  ay1   =      ay1+(ak11+2*ak21+2*ak31+ak41)/6

  ; R-K Section 2
  ak12  =      ih*((ay1-ay2)*kfc)
  ak22  =      ih*((ay1-(ay2+.5*ak12))*kfc)
  ak32  =      ih*((ay1-(ay2+.5*ak22))*kfc)
  ak42  =      ih*((ay1-(ay2+ak32))*kfc)
  ay2   =      ay2+(ak12+2*ak22+2*ak32+ak42)/6

  ; Pentic bounce equation
  ax3    =      -.1*ay*kpb
  aaxs   =      (ax3*ax3*ax3*ax3*ax3+ay2)*1000*kpa     ; Update acceleration

  ; R-K Section 3
  ak13  =      ih*((ay2-ay3)*kfc+aaxs)
  ak23  =      ih*((ay2-(ay3+.5*ak13))*kfc+aaxs)
  ak33  =      ih*((ay2-(ay3+.5*ak23))*kfc+aaxs)
  ak43  =      ih*((ay2-(ay3+ak33))*kfc+aaxs)
  ay3   =      ay3+(ak13+2*ak23+2*ak33+ak43)/6

  ; R-K Section 4
  ak14  =      ih*((ay3-ay)*kfc)
  ak24  =      ih*((ay3-(ay+.5*ak14))*kfc)
  ak34  =      ih*((ay3-(ay+.5*ak24))*kfc)
  ak44  =      ih*((ay3-(ay+ak34))*kfc)
  ay    =      ay+(ak14+2*ak24+2*ak34+ak44)/6

aout   =      ay*iamp*kdclck*.07 ; Apply amp envelope and declick

       outs   aout*ipanl, aout*ipanr  ; Output the sound

       endin
```

-- Hans Mikelson <hljmm@charter.net>

All-Pass Filters, a good explanation (click this to go back to the index)

Type : information

References : Posted by Olli Niemitalo

Linked file : filters002.txt (this linked file is included below)

Linked files

All-Pass Filters

A true allpass is not just:

```
|
|
|     |
|     |     .
|     |     |     |     .
|_____|_____|_____|_____|_____|_____|_____.
```

Instead, the first peak is negative and has a carefully chosen height.

```
      |
      |     |     .
 _____|_____|_____|_____|_____|_____._____.
|
|
|
|
|
```

This can be achieved by adding scaled in(T) to the scaled output of:

out(T) = in(T) + out(T-delay)*gain

Choosing the scalings right... Looking at the critical frequencies, those that go A) a number of full cycle per peak, and those that go B) a number of full cycles and a half-cycle per peak, we can set the constraints to eliminate cancellation:

A) The sum of all peaks must be 1.

B) The sum of odd peaks minus the sum of even peaks (including the negative peak which we give number zero) must be 1.

Let's get into business now that we know what we want. We call the amplitude of the negative peak "a" and the amplitude of the first postive peak "b". The ratio between adjacent positive peaks, the feedback gain, is denoted by "g".

A) The sum of positive peaks is a geometric series and simplifies to b/(1-g). Our first constraint becomes:

a + b/(1-g) = 1.

B) Using similar math... The sum of odd peaks is b/(1-g^2). The sum of even peaks is a + b*g/(1-g^2). So the second constraint is formed:

b/(1-g^2) - (a + b*g/(1-g^2)) = 1.

Solving the two equations we get:

a = -g
b = 1-g^2

Here i had a GREAT phewwww feeling of relief looking at
http://harmony-central.com/Effects/Articles/Reverb/allpass.html

Choosing g is up to you. You can even make it negative, just remember to keep |g| < 1. Gosh, perhaps complex g could be used to create a pulsating response (forget i said that)!

We can write the allpass routine in a programmer-wise pleasant way, still preserving ease of mathematical analysis:

```
mid(T) = in(T) + g*mid(T-delay);
out(T) = (1/g-g)*mid(T) - (1/g)*in(T);
```

That can be thought of as two filters put in serial and a third one in parallel with them. The first of the two serial ones is the delay with feedback. The other "filters" are just different gains.

For the first code line, the frequency response is the usual set peaks:

```
         1
--------------------
1 - g e^(-i w delay)
```

Adding the mixing on the second code line gives the flat-magnitude frequency response of the whole allpass system:

```
|       1/g - g              |
| -------------------- - 1/g | = 1
| 1 - g e^(-i w delay)       |
```

The phase response is a wavy one: (formula not double-checked!)

```
        (g^2-1) sin(delay w)
atan -------------------------
     (g^2+1) cos(delay w) - 2g
```

I hope this cleared things out - and that there aren't fatal errors! :)

-- Olli Niemitalo <oniemita@mail.student.oulu.fi>

Reprise des calculs (T = delay) :

```
       1/g - g        1
H = --------------- - -
    1 - g*exp(-iwT)   g

    1 - g^2 - (1 - g*exp(-iwT))
H = ---------------------------
       (1 - g*exp(-iwT)) * g

    -g + exp(-iwT)
H = ----------------
    1 - g*exp(-iwT)

                1 - g*exp(iwT)
H = exp(-iwT) * ---------------
                1 - g*exp(-iwT)
```

Le numerateur et le denominateur sont conjugues. d'ou :

|H| = 1

et

```
                              g * sin(wT)
arg (H) = -wT - 2 * arctan ---------------
                           1 - g * cos(wT)
```

Ce dephasage est le meme que celui trouve par Olli mais a l'avantage de separer le retard global et le dephasage.

-- Laurent de Soras <ldesoras@club-internet.fr>

More generally (in fact, maximally generally) you will get an all-pass response with any transfer function of the form :

```
           z^-n * sum (a(i) * z^i)
H(z) = b * ------------------------
           sum (conj (a(i)) * z^-i)
```

where |b| = 1 and of course n should be large enough that your filter is causal.

-- Frederick Umminger <fumminger@my-Deja.com>

Biquad C code (click this to go back to the index)

References : Posted by Tom St Denis

Linked file : biquad.c (this linked file is included below)

Notes :
Implementation of the RBJ cookbook, in C.

Linked files
```
/* Simple implementation of Biquad filters -- Tom St Denis
 *
 * Based on the work

Cookbook formulae for audio EQ biquad filter coefficients
---------------------------------------------------------
by Robert Bristow-Johnson, pbjrbj@viconet.com  a.k.a. robert@audioheads.com

 * Available on the web at

http://www.smartelectronix.com/musicdsp/text/filters005.txt

 * Enjoy.
 *
 * This work is hereby placed in the public domain for all purposes, whether
 * commercial, free [as in speech] or educational, etc.  Use the code and please
 * give me credit if you wish.
 *
 * Tom St Denis -- http://tomstdenis.home.dhs.org
*/

/* this would be biquad.h */
#include <math.h>
#include <stdlib.h>

#ifndef M_LN2
#define M_LN2    0.69314718055994530942
#endif

#ifndef M_PI
#define M_PI  3.14159265358979323846
#endif

/* whatever sample type you want */
typedef double smp_type;

/* this holds the data required to update samples thru a filter */
typedef struct {
    smp_type a0, a1, a2, a3, a4;
    smp_type x1, x2, y1, y2;
}
biquad;

extern smp_type BiQuad(smp_type sample, biquad * b);
extern biquad *BiQuad_new(int type, smp_type dbGain, /* gain of filter */
                          smp_type freq,             /* center frequency */
                          smp_type srate,            /* sampling rate */
                          smp_type bandwidth);       /* bandwidth in octaves */

/* filter types */
enum {
    LPF, /* low pass filter */
    HPF, /* High pass filter */
    BPF, /* band pass filter */
    NOTCH, /* Notch Filter */
    PEQ, /* Peaking band EQ filter */
    LSH, /* Low shelf filter */
    HSH /* High shelf filter */
};

/* Below this would be biquad.c */
/* Computes a BiQuad filter on a sample */
smp_type BiQuad(smp_type sample, biquad * b)
{
    smp_type result;

    /* compute result */
    result = b->a0 * sample + b->a1 * b->x1 + b->a2 * b->x2 -
        b->a3 * b->y1 - b->a4 * b->y2;

    /* shift x1 to x2, sample to x1 */
    b->x2 = b->x1;
```

```
    b->x1 = sample;

    /* shift y1 to y2, result to y1 */
    b->y2 = b->y1;
    b->y1 = result;

    return result;
}

/* sets up a BiQuad Filter */
biquad *BiQuad_new(int type, smp_type dbGain, smp_type freq,
smp_type srate, smp_type bandwidth)
{
    biquad *b;
    smp_type A, omega, sn, cs, alpha, beta;
    smp_type a0, a1, a2, b0, b1, b2;

    b = malloc(sizeof(biquad));
    if (b == NULL)
        return NULL;

    /* setup variables */
    A = pow(10, dbGain /40);
    omega = 2 * M_PI * freq /srate;
    sn = sin(omega);
    cs = cos(omega);
    alpha = sn * sinh(M_LN2 /2 * bandwidth * omega /sn);
    beta = sqrt(A + A);

    switch (type) {
    case LPF:
        b0 = (1 - cs) /2;
        b1 = 1 - cs;
        b2 = (1 - cs) /2;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case HPF:
        b0 = (1 + cs) /2;
        b1 = -(1 + cs);
        b2 = (1 + cs) /2;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case BPF:
        b0 = alpha;
        b1 = 0;
        b2 = -alpha;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case NOTCH:
        b0 = 1;
        b1 = -2 * cs;
        b2 = 1;
        a0 = 1 + alpha;
        a1 = -2 * cs;
        a2 = 1 - alpha;
        break;
    case PEQ:
        b0 = 1 + (alpha * A);
        b1 = -2 * cs;
        b2 = 1 - (alpha * A);
        a0 = 1 + (alpha /A);
        a1 = -2 * cs;
        a2 = 1 - (alpha /A);
        break;
    case LSH:
        b0 = A * ((A + 1) - (A - 1) * cs + beta * sn);
        b1 = 2 * A * ((A - 1) - (A + 1) * cs);
        b2 = A * ((A + 1) - (A - 1) * cs - beta * sn);
        a0 = (A + 1) + (A - 1) * cs + beta * sn;
        a1 = -2 * ((A - 1) + (A + 1) * cs);
        a2 = (A + 1) + (A - 1) * cs - beta * sn;
        break;
    case HSH:
        b0 = A * ((A + 1) + (A - 1) * cs + beta * sn);
        b1 = -2 * A * ((A - 1) + (A + 1) * cs);
        b2 = A * ((A + 1) + (A - 1) * cs - beta * sn);
        a0 = (A + 1) - (A - 1) * cs + beta * sn;
```

```
        a1 = 2 * ((A - 1) - (A + 1) * cs);
        a2 = (A + 1) - (A - 1) * cs - beta * sn;
        break;
    default:
        free(b);
        return NULL;
    }

    /* precompute the coefficients */
    b->a0 = b0 /a0;
    b->a1 = b1 /a0;
    b->a2 = b2 /a0;
    b->a3 = a1 /a0;
    b->a4 = a2 /a0;

    /* zero initial samples */
    b->x1 = b->x2 = 0;
    b->y1 = b->y2 = 0;

    return b;
}
/* crc==3062280887, version==4, Sat Jul  7 00:03:23 2001 */
```

<u>C++ class implementation of RBJ Filters</u> (click this to go back to the index)

<u>References</u> : Posted by arguru[AT]smartelectronix[DOT]com

<u>Linked file</u> : CFxRbjFilter.h (this linked file is included below)

<u>Notes</u> :
[WARNING: This code is not FPU undernormalization safe!]

<u>Linked files</u>

```
class CFxRbjFilter
{
public:

 CFxRbjFilter()
 {
  // reset filter coeffs
  b0a0=b1a0=b2a0=a1a0=a2a0=0.0;

  // reset in/out history
  ou1=ou2=in1=in2=0.0f;
 };

 float filter(float in0)
 {
  // filter
  float const yn = b0a0*in0 + b1a0*in1 + b2a0*in2 - a1a0*ou1 - a2a0*ou2;

  // push in/out buffers
  in2=in1;
  in1=in0;
  ou2=ou1;
  ou1=yn;

  // return output
  return yn;
 };

 void calc_filter_coeffs(int const type,double const frequency,double const sample_rate,double
const q,double const db_gain,bool q_is_bandwidth)
 {
  // temp pi
  double const temp_pi=3.1415926535897932384626433832795;

  // temp coef vars
  double alpha,a0,a1,a2,b0,b1,b2;

  // peaking, lowshelf and hishelf
  if(type>=6)
  {
   double const A  = pow(10.0,(db_gain/40.0));
   double const omega = 2.0*temp_pi*frequency/sample_rate;
   double const tsin = sin(omega);
   double const tcos = cos(omega);

   if(q_is_bandwidth)
   alpha=tsin*sinh(log(2.0)/2.0*q*omega/tsin);
   else
   alpha=tsin/(2.0*q);

   double const beta = sqrt(A)/q;

   // peaking
   if(type==6)
   {
    b0=float(1.0+alpha*A);
    b1=float(-2.0*tcos);
    b2=float(1.0-alpha*A);
    a0=float(1.0+alpha/A);
    a1=float(-2.0*tcos);
    a2=float(1.0-alpha/A);
   }

   // lowshelf
   if(type==7)
   {
    b0=float(A*((A+1.0)-(A-1.0)*tcos+beta*tsin));
    b1=float(2.0*A*((A-1.0)-(A+1.0)*tcos));
    b2=float(A*((A+1.0)-(A-1.0)*tcos-beta*tsin));
    a0=float((A+1.0)+(A-1.0)*tcos+beta*tsin);
    a1=float(-2.0*((A-1.0)+(A+1.0)*tcos));
    a2=float((A+1.0)+(A-1.0)*tcos-beta*tsin);
```

```
  }

  // hishelf
  if(type==8)
  {
   b0=float(A*((A+1.0)+(A-1.0)*tcos+beta*tsin));
   b1=float(-2.0*A*((A-1.0)+(A+1.0)*tcos));
   b2=float(A*((A+1.0)+(A-1.0)*tcos-beta*tsin));
   a0=float((A+1.0)-(A-1.0)*tcos+beta*tsin);
   a1=float(2.0*((A-1.0)-(A+1.0)*tcos));
   a2=float((A+1.0)-(A-1.0)*tcos-beta*tsin);
  }
 }
 else
 {
  // other filters
  double const omega = 2.0*temp_pi*frequency/sample_rate;
  double const tsin = sin(omega);
  double const tcos = cos(omega);

  if(q_is_bandwidth)
  alpha=tsin*sinh(log(2.0)/2.0*q*omega/tsin);
  else
  alpha=tsin/(2.0*q);


  // lowpass
  if(type==0)
  {
   b0=(1.0-tcos)/2.0;
   b1=1.0-tcos;
   b2=(1.0-tcos)/2.0;
   a0=1.0+alpha;
   a1=-2.0*tcos;
   a2=1.0-alpha;
  }

  // hipass
  if(type==1)
  {
   b0=(1.0+tcos)/2.0;
   b1=-(1.0+tcos);
   b2=(1.0+tcos)/2.0;
   a0=1.0+ alpha;
   a1=-2.0*tcos;
   a2=1.0-alpha;
  }

  // bandpass csg
  if(type==2)
  {
   b0=tsin/2.0;
   b1=0.0;
      b2=-tsin/2;
   a0=1.0+alpha;
   a1=-2.0*tcos;
   a2=1.0-alpha;
  }

  // bandpass czpg
  if(type==3)
  {
   b0=alpha;
   b1=0.0;
   b2=-alpha;
   a0=1.0+alpha;
   a1=-2.0*tcos;
   a2=1.0-alpha;
  }

  // notch
  if(type==4)
  {
   b0=1.0;
   b1=-2.0*tcos;
   b2=1.0;
   a0=1.0+alpha;
   a1=-2.0*tcos;
   a2=1.0-alpha;
  }

  // allpass
  if(type==5)
```

```
    {
     b0=1.0-alpha;
     b1=-2.0*tcos;
     b2=1.0+alpha;
     a0=1.0+alpha;
     a1=-2.0*tcos;
     a2=1.0-alpha;
    }
   }

   // set filter coeffs
   b0a0=float(b0/a0);
   b1a0=float(b1/a0);
   b2a0=float(b2/a0);
   a1a0=float(a1/a0);
   a2a0=float(a2/a0);
  };

private:

 // filter coeffs
 float b0a0,b1a0,b2a0,a1a0,a2a0;

 // in/out history
 float ou1,ou2,in1,in2;
};
```

Cascaded resonant lp/hp filter (click this to go back to the index)

Type : lp+hp

References : Posted by tobybear[AT]web[DOT]de

Notes :
// Cascaded resonant lowpass/hipass combi-filter
// The original source for this filter is from Paul Kellet from
// the archive. This is a cascaded version in Delphi where the
// output of the lowpass is fed into the highpass filter.
// Cutoff frequencies are in the range of 0<=x<1 which maps to
// 0..nyquist frequency

// input variables are:
// cut_lp: cutoff frequency of the lowpass (0..1)
// cut_hp: cutoff frequency of the hipass (0..1)
// res_lp: resonance of the lowpass (0..1)
// res_hp: resonance of the hipass (0..1)

Code :
```
var n1,n2,n3,n4:single; // filter delay, init these with 0!
    fb_lp,fb_hp:single; // storage for calculated feedback
const p4=1.0e-24; // Pentium 4 denormal problem elimination

function dofilter(inp,cut_lp,res_lp,cut_hp,res_hp:single):single;
begin
 fb_lp:=res_lp+res_lp/(1-cut_lp);
 fb_hp:=res_hp+res_hp/(1-cut_lp);
 n1:=n1+cut_lp*(inp-n1+fb_lp*(n1-n2))+p4;
 n2:=n2+cut_lp*(n1-n2);
 n3:=n3+cut_hp*(n2-n3+fb_hp*(n3-n4))+p4;
 n4:=n4+cut_hp*(n3-n4);
 result:=i-n4;
end;
```

Comments
from : office@hermannseib.com
comment : I guess the last line should read
result:=inp-n4;

Right?

Bye,

Hermann

from : couriervst@hotmail.com
comment : excuse me which type is? 6db/oct or 12 or what?

thanks

from : christianalthaus@gmx.de
comment : result := n2-n4

:)

DC filter (click this to go back to the index)

Type : 1-pole/1-zero DC filter

References : Posted by andy[DOT]rossol[AT]bluewin[DOT]ch

Notes :
This is based on code found in the document:
"Introduction to Digital Filters (DRAFT)"
Julius O. Smith III (jos@ccrma.stanford.edu)
(http://www-ccrma.stanford.edu/~jos/filters/)

Some audio algorithms (asymmetric waveshaping, cascaded filters, ...) can produce DC offset. This offset can accumulate and reduce the signal/noise ratio.

So, how to fix it? The example code from Julius O. Smith's document is:
...
y(n) = x(n) - x(n-1) + R * y(n-1)
// "R" between 0.9 .. 1
// n=current (n-1)=previous in/out value
...
"R" depends on sampling rate and the low frequency point. Do not set "R" to a fixed value (e.g. 0.99) if you don't know the
sample rate. Instead set R to:
(-3dB @ 40Hz): R = 1-(250/samplerate)
(-3dB @ 30Hz): R = 1-(190/samplerate)
(-3dB @ 20Hz): R = 1-(126/samplerate)

Comments
from : andy[DOT]rossol[AT]bluewin[DOT]ch
comment : I just received a mail from a musicdsp reader:

'How to calculate "R" for a given (-3dB) low frequency point?'

R = 1 - (pi*2 * frequency /samplerate)

(pi=3.14159265358979)

from : rbj@surfglobal.net
comment : particularly if fixed-point arithmetic is used, this simple high-pass filter can create it's own DC offset because of limit-cycles. to cure that look at

http://www.dspguru.com/comp.dsp/tricks/alg/dc_block.htm

this trick uses the concept of "noise-shaping" to prevent DC in any limit-cycles.

r b-j

Digital RIAA equalization filter coefficients (click this to go back to the index)

Type : RIAA

References : Posted by Frederick Umminger

Notes :
Use at your own risk. Confirm correctness before using. Don't assume I didn't goof something up.

-Frederick Umminger

Code :
```
The "turntable-input software" thread inspired me to generate some coefficients for a digital RIAA
equalization filter. These coefficients were found by matching the magnitude response of the s-domain transfer
function using some proprietary Matlab scripts. The phase response may or may not be totally whacked.

The s-domain transfer function is

R3(1+R1*C1*s)(1+R2*C2*s)/(R1(1+R2*C2*s) + R2(1+R1*C1*s) + R3(1+R1*C1*s)(1+R2*C2*s))

where

R1 = 883.3k
R2 = 75k
R3 = 604
C1 = 3.6n
C2 = 1n

This is based on the reference circuit found in http://www.hagtech.com/pdf/riaa.pdf

The coefficients of the digital transfer function b(z^-1)/a(z^-1) in descending powers of z, are:

44.1kHz
b = [ 0.02675918611906  -0.04592084787595   0.01921229297239]
a = [ 1.00000000000000  -0.73845850035973  -0.17951755477430]
error +/- 0.25dB

48kHz
b = [  0.02675918611906  -0.04592084787595   0.01921229297239]
a = [  1.00000000000000  -0.73845850035973  -0.17951755477430]
error +/- 0.15dB

88.2kHz
b = [  0.04872204977233  -0.09076930609195   0.04202280710877]
a = [ 1.00000000000000  -0.85197860443215  -0.10921171201431]
error +/- 0.01dB


96kHz
b = [ 0.05265477122714  -0.09864197097385   0.04596474352090  ]
a = [  1.00000000000000  -0.85835597216218  -0.10600020417219 ]
error +/- 0.006dB
```

Formant filter (click this to go back to the index)

References : Posted by Alex

Code :
```
/*
Public source code by alex@smartelectronix.com
Simple example of implementation of formant filter
Vowelnum can be 0,1,2,3,4 <=> A,E,I,O,U
Good for spectral rich input like saw or square
*/
//-------------------------------------------------------------VOWEL COEFFICIENTS
const double coeff[5][11]= {
{ 8.11044e-06,
8.943665402, -36.83889529, 92.01697887, -154.337906, 181.6233289,
-151.8651235,   89.09614114, -35.10298511, 8.388101016, -0.923313471  ///A
},
{4.36215e-06,
8.90438318, -36.55179099, 91.05750846, -152.422234, 179.1170248,  ///E
-149.6496211,87.78352223, -34.60687431, 8.282228154, -0.914150747
},
{ 3.33819e-06,
8.893102966, -36.49532826, 90.96543286, -152.4545478, 179.4835618,
-150.315433, 88.43409371, -34.98612086, 8.407803364, -0.932568035  ///I
},
{1.13572e-06,
8.994734087, -37.2084849, 93.22900521, -156.6929844, 184.596544,   ///O
-154.3755513, 90.49663749, -35.58964535, 8.478996281, -0.929252233
},
{4.09431e-07,
8.997322763, -37.20218544, 93.11385476, -156.2530937, 183.7080141,  ///U
-153.2631681, 89.59539726, -35.12454591, 8.338655623, -0.910251753
}
};
//---------------------------------------------------------------------------------
static double memory[10]={0,0,0,0,0,0,0,0,0,0};
//---------------------------------------------------------------------------------
float formant_filter(float *in, int vowelnum)
{
      res= (float) ( coeff[vowelnum][0]  *in +
      coeff[vowelnum][1]  *memory[0] +
      coeff[vowelnum][2]  *memory[1] +
      coeff[vowelnum][3]  *memory[2] +
      coeff[vowelnum][4]  *memory[3] +
      coeff[vowelnum][5]  *memory[4] +
      coeff[vowelnum][6]  *memory[5] +
      coeff[vowelnum][7]  *memory[6] +
      coeff[vowelnum][8]  *memory[7] +
      coeff[vowelnum][9]  *memory[8] +
      coeff[vowelnum][10] *memory[9] );

memory[9]= memory[8];
memory[8]= memory[7];
memory[7]= memory[6];
memory[6]= memory[5];
memory[5]= memory[4];
memory[4]= memory[3];
memory[3]= memory[2];
memory[2]= memory[1];
memory[1]= memory[0];
memory[0]=(double) res;
return res;
}
```

Comments
  from : rhettanderson@yahoo.com
  comment : Where did the coefficients come from? Do they relate to frequencies somehow? Are they male or female? Etc.

  from : el98shn@ing.umu.se
  comment : And are the coeffiecients for 44k1hz?
/stefancrs

  from : meeloo@meeloo.net
  comment : It seem to be ok at 44KHz although I get quite lot of distortion with this filter.
There are typos in the given code too, the correct version looks like this i think:
```
float formant_filter(float *in, int vowelnum)
{
  float res= (float) ( coeff[vowelnum][0]* (*in) +
  coeff[vowelnum][1] *memory[0] +
  coeff[vowelnum][2] *memory[1] +
  coeff[vowelnum][3] *memory[2] +
  coeff[vowelnum][4] *memory[3] +
  coeff[vowelnum][5] *memory[4] +
  coeff[vowelnum][6] *memory[5] +
  coeff[vowelnum][7] *memory[6] +
  coeff[vowelnum][8] *memory[7] +
  coeff[vowelnum][9] *memory[8] +
  coeff[vowelnum][10] *memory[9] );
```

...

(missing type and asterisk in the first calc line ;).

I tried morphing from one vowel to another and it works ok except in between 'A' and 'U' as I get a lot of distortion and sometime (depending on the signal) the filter goes into auto-oscilation.

Sebastien Metrot

from : larsby@elak.org
comment : How did you get the coeffiecients?

Did I miss something?

/Larsby

from : stefan.hallen@dice.se
comment : Yeah, morphing lineary between the coefficients works just fine. The distortion I only get when not lowering the amplitude of the input. So I lower it :)

Larsby, you can approximate filter curves quite easily, check your dsp literature :)

from : alex@smartelectronix.com
comment : Correct, it is for sampling rate of 44kHz.
It supposed to be female (soprano), approximated with its five formants.

--Alex.

from : ruiner33@hotmail.com
comment : Can you tell us how you calculated the coefficients?

from : antiprosynthesis@hotmail.com
comment : The distorting/sharp A vowel can be toned down easy by just changing the first coeff from 8.11044e-06 to 3.11044e-06. Sounds much better that way.

Karlsen (click this to go back to the index)

Type : 24-dB (4-pole) lowpass

References : Posted by Best Regards,Ove Karlsen

Notes :
There's really not much voodoo going on in the filter itself, it's a simple as possible:

pole1 = (in * frequency) + (pole1 * (1 - frequency));

Most of you can probably understand that math, it's very similar to how an analog condenser works.
Although, I did have to do some JuJu to add resonance to it.
While studing the other filters, I found that the feedback phase is very important to how the overall resonance level will be, and so I made a dynamic feedback path, and constant Q approximation by manipulation of the feedback phase.
A bonus with this filter, is that you can "overdrive" it... Try high input levels..

Code :
```
// Karlsen 24dB Filter by Ove Karlsen / Synergy-7 in the year 2003.
// b_f = frequency 0..1
// b_q = resonance 0..50
// b_in = input
// to do bandpass, subtract poles from eachother, highpass subtract with input.


   float b_inSH = b_in // before the while statement.

 while (b_oversample < 2) {      //2x oversampling (@44.1khz)
  float prevfp;
  prevfp = b_fp;
  if (prevfp > 1) {prevfp = 1;}     // Q-limiter

  b_fp = (b_fp * 0.418) + ((b_q * pole4) * 0.582);  // dynamic feedback
  float intfp;
  intfp = (b_fp * 0.36) + (prevfp * 0.64);   // feedback phase
  b_in = b_inSH - intfp;      // inverted feedback

  pole1 = (b_in   * b_f) + (pole1 * (1 - b_f));   // pole 1
  if (pole1 > 1) {pole1 = 1;} else if (pole1 < -1) {pole1 = -1;}  // pole 1 clipping
  pole2 = (pole1   * b_f) + (pole2 * (1 - b_f));   // pole 2
  pole3 = (pole2   * b_f) + (pole3 * (1 - b_f));   // pole 3
  pole4 = (pole3   * b_f) + (pole4 * (1 - b_f));   // pole 4

  b_oversample++;
  }
 lowpassout = b_in;
```

Comments
from : ove@synergy-7.com
comment : Hi.
Seems to be a slight typo in my code.
lowpassout = pole4; // ofcourse :)

Best Regards,
Ove Karlsen

from : matt at ahsodit dot com
comment : Hi Ove, we spoke once on the #AROS IRC channel... I'm trying to put this code into a filter object, but I'm wandering what datatype the input and output should be?

I'm processing my audio data in packets of 8000 signed words (16 bits) at a time. can I put one audio sample words into this function? Since it seems to require a floating point input!

Thanks

Lowpass filter for parameter edge filtering (click this to go back to the index)

References : Olli Niemitalo

Linked file : filter001.gif

Notes :
use this filter to smooth sudden parameter changes
(see linkfile!)

Code :
```
/* - Three one-poles combined in parallel
 * - Output stays within input limits
 * - 18 dB/oct (approx) frequency response rolloff
 * - Quite fast, 2x3 parallel multiplications/sample, no internal buffers
 * - Time-scalable, allowing use with different samplerates
 * - Impulse and edge responses have continuous differential
 * - Requires high internal numerical precision
 */
{
        /* Parameters */
        // Number of samples from start of edge to halfway to new value
        const double        scale = 100;
        // 0 < Smoothness < 1. High is better, but may cause precision problems
        const double        smoothness = 0.999;

        /* Precalc variables */
        double                a = 1.0-(2.4/scale); // Could also be set directly
        double                b = smoothness;      //          -"-
        double                acoef = a;
        double                bcoef = a*b;
        double                ccoef = a*b*b;
        double                mastergain = 1.0 / (-1.0/(log(a)+2.0*log(b))+2.0/
                        (log(a)+log(b))-1.0/log(a));
        double                again = mastergain;
        double                bgain = mastergain * (log(a*b*b)*(log(a)-log(a*b)) /
                            ((log(a*b*b)-log(a*b))*log(a*b))
                            - log(a)/log(a*b));
        double                cgain = mastergain * (-(log(a)-log(a*b)) /
                        (log(a*b*b)-log(a*b)));

        /* Runtime variables */
        long                streamofs;
        double                areg = 0;
        double                breg = 0;
        double                creg = 0;

        /* Main loop */
        for (streamofs = 0; streamofs < streamsize; streamofs++)
        {
                /* Update filters */
                areg = acoef * areg + fromstream [streamofs];
                breg = bcoef * breg + fromstream [streamofs];
                creg = ccoef * creg + fromstream [streamofs];

                /* Combine filters in parallel */
                long                temp =   again * areg
                                       + bgain * breg
                                       + cgain * creg;

                /* Check clipping */
                if (temp > 32767)
                {
                        temp = 32767;
                }
                else if (temp < -32768)
                {
                        temp = -32768;
                }

                /* Store new value */
                tostream [streamofs] = temp;
        }
}
```

LP and HP filter (click this to go back to the index)

Type : biquad, tweaked butterworth

References : Posted by Patrice Tarrabia

Code :
```
r  = rez amount, from sqrt(2) to ~ 0.1
f  = cutoff frequency
(from ~0 Hz to SampleRate/2 - though many
synths seem to filter only  up to SampleRate/4)

The filter algo:
out(n) = a1 * in + a2 * in(n-1) + a3 * in(n-2) - b1*out(n-1) - b2*out(n-2)

Lowpass:
      c = 1.0 / tan(pi * f / sample_rate);

      a1 = 1.0 / ( 1.0 + r * c + c * c);
      a2 = 2* a1;
      a3 = a1;
      b1 = 2.0 * ( 1.0 - c*c) * a1;
      b2 = ( 1.0 - r * c + c * c) * a1;

Hipass:
      c = tan(pi * f / sample_rate);

      a1 = 1.0 / ( 1.0 + r * c + c * c);
      a2 = -2*a1;
      a3 = a1;
      b1 = 2.0 * ( c*c - 1.0) * a1;
      b2 = ( 1.0 - r * c + c * c) * a1;
```

Comments

from : andy_rossol@hotmail.com

comment : Ok, the filter works, but how to use the resonance parameter (r)? The range from sqrt(2)-lowest to 0.1 (highest res.) is Ok for a LP with Cutoff > 3 or 4 KHz, but for lower cutoff frequencies and higher res you will get values much greater than 1! (And this means clipping like hell)

So, has anybody calculated better parameters (for r, b1, b2)?

from : kainhart@hotmail.com

comment : Below is my attempt to implement the above lowpass filter in c#. I'm just a beginner at this so it's probably something that I've messed up. If anybody can offer a suggestion of what I may be doing wrong please help. I'm getting a bunch of stable staticky noise as my output of this filter currently.

from : kainhart@hotmail.com

comment : public class LowPassFilter
```
{
    /// <summary>
    /// rez amount, from sqrt(2) to ~ 0.1
    /// </summary>
    float r;
    /// <summary>
    /// cutoff frequency
    /// (from ~0 Hz to SampleRate/2 - though many
    /// synths seem to filter only up to SampleRate/4)
    ///</summary>
    float f;
    float c;

    float a1;
    float a2;
    float a3;
    float b1;
    float b2;

//  float in0 = 0;
    float in1 = 0;
    float in2 = 0;

//  float out0;
    float out1 = 0;
    float out2 = 0;

    private int _SampleRate;

    public LowPassFilter(int sampleRate)
    {
        _SampleRate = sampleRate;

//      SetParams(_SampleRate / 2f, 0.1f);
        SetParams(_SampleRate / 8f, 1f);
    }
```

```
    public float Process(float input)
    {
        float output = a1 * input +
                 a2 * in1 +
                 a3 * in2 -
                 b1 * out1 -
                 b2 * out2;

        in2 = in1;
        in1 = input;

        out2 = out1;
        out1 = output;

        Console.WriteLine(input + ", " + output);

        return output;
    }
```

from : kainhart@hotmail.com
comment :

```
/// <summary>
    ///
    /// </summary>
    public float CutoffFrequency
    {
        set
        {
            f = value;
            c = (float) (1.0f / Math.Tan(Math.PI * f / _SampleRate));
            SetParams();
        }
        get
        {
            return f;
        }
    }

    /// <summary>
    ///
    /// </summary>
    public float Resonance
    {
        set
        {
            r = value;
            SetParams();
        }
        get
        {
            return r;
        }
    }

    public void SetParams(float cutoffFrequency, float resonance)
    {
        r = resonance;
        CutoffFrequency = cutoffFrequency;
    }

    /// <summary>
    /// TODO rename
    /// </summary>
    /// <param name="c"></param>
    /// <param name="resonance"></param>
    private void SetParams()
    {
        a1 = 1f / (1f + r*c + c*c);
        a2 = 2 * a1;
        a3 = a1;
        b1 = 2f * (1f - c*c) * a1;
        b2 = (1f - r*c + c*c) * a1;
    }
}
```

from : kainhart@hotmail.com
comment : Nevermind I think I solved my problem. I was missing parens around the coefficients and the variables ...(a1 * input)...

from : kainhart[AT]hotmail.com
comment : After implementing the lowpass algorithm I get a loud ringing noise on some frequencies both high and low. Any ideas?

Moog VCF (click this to go back to the index)

Type : 24db resonant lowpass

References : CSound source code, Stilson/Smith CCRMA paper.

Notes :
Digital approximation of Moog VCF. Fairly easy to calculate coefficients, fairly easy to process algorithm, good sound.

Code :
```
//Init
cutoff = cutoff freq in Hz
fs = sampling frequency //(e.g. 44100Hz)
res = resonance [0 - 1] //(minimum - maximum)

f = 2 * cutoff / fs; //[0 - 1]
k = 3.6*f - 1.6*f*f -1; //(Empirical tunning)
p = (k+1)*0.5;
scale = e^((1-p)*1.386249;
r = res*scale;
y4 = output;

y1=y2=y3=y4=oldx=oldy1=oldy2=oldy3=0;

//Loop
//--Inverted feed back for corner peaking
x = input - r*y4;

//Four cascaded onepole filters (bilinear transform)
y1=x*p + oldx*p - k*y1;
y2=y1*p+oldy1*p - k*y2;
y3=y2*p+oldy2*p - k*y3;
y4=y3*p+oldy3*p - k*y4;

//Clipper band limited sigmoid
y4 = y4 - (y4^3)/6;

oldx = x;
oldy1 = y1;
oldy2 = y2;
oldy3 = y3;
```

Moog VCF, variation 1 (click this to go back to the index)

Type : 24db resonant lowpass

References : CSound source code, Stilson/Smith CCRMA paper., Paul Kellett version

Notes :
The second "q =" line previously used exp() - I'm not sure if what I've done is any faster, but this line needs playing with anyway as it controls which frequencies will self-oscillate. I
think it could be tweaked to sound better than it currently does.

Highpass / Bandpass :

They are only 6dB/oct, but still seem musically useful - the 'fruity' sound of the 24dB/oct lowpass is retained.

Code :
```
// Moog 24 dB/oct resonant lowpass VCF
// References: CSound source code, Stilson/Smith CCRMA paper.
// Modified by paul.kellett@maxim.abel.co.uk July 2000

  float f, p, q;             //filter coefficients
  float b0, b1, b2, b3, b4;  //filter buffers (beware denormals!)
  float t1, t2;              //temporary buffers

// Set coefficients given frequency & resonance [0.0...1.0]

  q = 1.0f - frequency;
  p = frequency + 0.8f * frequency * q;
  f = p + p - 1.0f;
  q = resonance * (1.0f + 0.5f * q * (1.0f - q + 5.6f * q * q));

// Filter (in [-1.0...+1.0])

  in -= q * b4;                          //feedback
  t1 = b1;  b1 = (in + b0) * p - b1 * f;
  t2 = b2;  b2 = (b1 + t1) * p - b2 * f;
  t1 = b3;  b3 = (b2 + t2) * p - b3 * f;
            b4 = (b3 + t1) * p - b4 * f;
  b4 = b4 - b4 * b4 * b4 * 0.166667f;    //clipping
  b0 = in;

// Lowpass  output:  b4
// Highpass output:  in - b4;
// Bandpass output:  3.0f * (b3 - b4);
```

Moog VCF, variation 2 (click this to go back to the index)

Type : 24db resonant lowpass

References : CSound source code, Stilson/Smith CCRMA paper., Timo Tossavainen (?) version

Notes :
in[x] and out[x] are member variables, init to 0.0 the controls:

fc = cutoff, nearly linear [0,1] -> [0, fs/2]
res = resonance [0, 4] -> [no resonance, self-oscillation]

Code :
```
Tdouble MoogVCF::run(double input, double fc, double res)
{
  double f = fc * 1.16;
  double fb = res * (1.0 - 0.15 * f * f);
  input -= out4 * fb;
  input *= 0.35013 * (f*f)*(f*f);
  out1 = input + 0.3 * in1 + (1 - f) * out1; // Pole 1
  in1  = input;
  out2 = out1 + 0.3 * in2 + (1 - f) * out2;  // Pole 2
  in2  = out1;
  out3 = out2 + 0.3 * in3 + (1 - f) * out3;  // Pole 3
  in3  = out2;
  out4 = out3 + 0.3 * in4 + (1 - f) * out4;  // Pole 4
  in4  = out3;
  return out4;
}
```

Comments
from : askywhale@yahoo.fr
comment : This one works pretty well, thanks !

from : rdupres[AT]hotmail.com
comment : could somebody explain, what means this

input -= out4 * fb;
input *= 0.35013 * (f*f)*(f*f);

is "input-" and "input *" the name of an variable ??
or is this an Csound specific parameter ??
I want to translate this piece to Assemblercode
Robert Dupres

from : johanc@popbandetleif.cjb.net
comment : input is name of a variable with type double.

input -= out4 * fb;

is just a shorter way for writing:

input = input - out4 * fb;

and the *= operator is works similar:

input *= 0.35013 * (f*f)*(f*f);

is equal to

input = input * 0.35013 * (f*f)*(f*f);

/ Johan

Notch filter (click this to go back to the index)

Type : 2 poles 2 zeros IIR

References : Posted by Olli Niemitalo

Notes :
Creates a muted spot in the spectrum with adjustable steepness. A complex conjugate pair of zeros on the z- plane unit circle and neutralizing poles approaching at the same angles from inside the unit circle.

Code :
```
Parameters:
0 =< freq =< samplerate/2
0 =< q < 1 (The higher, the narrower)

AlgoAlgo=double pi = 3.141592654;
double sqrt2 = sqrt(2.0);

double freq = 2050; // Change! (zero & pole angle)
double q = 0.4;     // Change! (pole magnitude)

double z1x = cos(2*pi*freq/samplerate);
double a0a2 = (1-q)*(1-q)/(2*(fabs(z1x)+1)) + q;
double a1 = -2*z1x*a0a2;
double b1 = -2*z1x*q;
double b2 = q*q;
double reg0, reg1, reg2;

unsigned int streamofs;
reg1 = 0;
reg2 = 0;

/* Main loop */
for (streamofs = 0; streamofs < streamsize; streamofs++)
{
  reg0 = a0a2 * ((double)fromstream[streamofs]
                 + fromstream[streamofs+2])
       + a1 * fromstream[streamofs+1]
       - b1 * reg1
       - b2 * reg2;

  reg2 = reg1;
  reg1 = reg0;

  int temp = reg0;

  /* Check clipping */
  if (temp > 32767) {
    temp = 32767;
  } else if (temp < -32768) temp = -32768;

  /* Store new value */
  tostream[streamofs] = temp;
}
```

One pole LP and HP (click this to go back to the index)

References : Posted by Bram

Code :
```
LP:
recursion: tmp = (1-p)*in + p*tmp with output = tmp
coefficient: p = (2-cos(x)) - sqrt((2-cos(x))^2 - 1) with x = 2*pi*cutoff/samplerate
coeficient approximation: p = (1 - 2*cutoff/samplerate)^2

HP:
recursion: tmp = (p-1)*in - p*tmp with output = tmp
coefficient: p = (2+cos(x)) - sqrt((2+cos(x))^2 - 1) with x = 2*pi*cutoff/samplerate
coeficient approximation: p = (2*cutoff/samplerate)^2
```

One pole, one zero LP/HP (click this to go back to the index)

References : Posted by mistert[AT]inwind[DOT]it

Code :
```
void SetLPF(float fCut, float fSampling)
{
    float w = 2.0 * fSampling;
    float Norm;

    fCut *= 2.0F * PI;
    Norm = 1.0 / (fCut + w);
    b1 = (w - fCut) * Norm;
    a0 = a1 = fCut * Norm;
}

void SetHPF(float fCut, float fSampling)
{
    float w = 2.0 * fSampling;
    float Norm;

    fCut *= 2.0F * PI;
    Norm = 1.0 / (fCut + w);
    a0 = w * Norm;
    a1 = -a0;
    b1 = (w - fCut) * Norm;
}

Where
out[n] = in[n]*a0 + in[n-1]*a1 + out[n-1]*b1;
```

One zero, LP/HP (click this to go back to the index)

References : Posted by Bram

Notes :
LP is only 'valid' for cutoffs > samplerate/4
HP is only 'valid' for cutoffs < samplerate/4

Code :
```
theta = cutoff*2*pi / samplerate

LP:
H(z) = (1+p*z^(-1)) / (1+p)
out[i] = 1/(1+p) * in[i] + p/(1+p) * in[i-1];
p = (1-2*cos(theta)) - sqrt((1-2*cos(theta))^2 - 1)
Pi/2 < theta < Pi

HP:
H(z) = (1-p*z^(-1)) / (1+p)
out[i] = 1/(1+p) * in[i] - p/(1+p) * in[i-1];
p = (1+2*cos(theta)) - sqrt((1+2*cos(theta))^2 - 1)
0 < theta < Pi/2
```

Comments
from : newbie attack
comment : What is the implementation of z^(-1)?

Peak/Notch filter (click this to go back to the index)

Type : peak/notch

References : Posted by tobybear[AT]web[DOT]de

Notes :
// Peak/Notch filter
// I don't know anymore where this came from, just found it on
// my hard drive :-)
// Seems to be a peak/notch filter with adjustable slope
// steepness, though slope gets rather wide the lower the
// frequency is.
// "cut" and "steep" range is from 0..1
// Try to feed it with white noise, then the peak output does
// rather well eliminate all other frequencies except the given
// frequency in higher frequency ranges.

Code :
```
var f,r:single;
    outp,outp1,outp2:single; // init these with 0!
const p4=1.0e-24; // Pentium 4 denormal problem elimination

function PeakNotch(inp,cut,steep:single;ftype:integer):single;
begin
 r:=steep*0.99609375;
 f:=cos(pi*cut);
 a0:=(1-r)*sqrt(r*(r-4*(f*f)+2)+1);
 b1:=2*f*r;
 b2:=-(r*r);
 outp:=a0*inp+b1*outp1+b2*outp2+p4;
 outp2:=outp1;
 outp1:=outp;
 if ftype=0 then
  result:=outp //peak
 else
  result:=inp-outp; //notch
end;
```

Phase equalization (click this to go back to the index)

Type : Allpass

References : Posted by Uli the Grasso

Notes :
The idea is simple: One can equalize the phase response of a system, for example of a loudspeaker, by approximating its phase response by an FIR filter and then turn around the coefficients of the filter. At http://grassomusic.de/english/phaseeq.htm you find more info and an Octave script.

Comments
from : piem
comment : hi.
i couldn't find any script there...
cheers, piem

Pink noise filter (click this to go back to the index)

References : Posted by Paul Kellett

Linked file : pink.txt (this linked file is included below)

Notes :
(see linked file)

Linked files

```
Filter to make pink noise from white  (updated March 2000)
------------------------------------

This is an approximation to a -10dB/decade filter using a weighted sum
of first order filters. It is accurate to within +/-0.05dB above 9.2Hz
(44100Hz sampling rate). Unity gain is at Nyquist, but can be adjusted
by scaling the numbers at the end of each line.

If 'white' consists of uniform random numbers, such as those generated
by the rand() function, 'pink' will have an almost gaussian level
distribution.


  b0 = 0.99886 * b0 + white * 0.0555179;
  b1 = 0.99332 * b1 + white * 0.0750759;
  b2 = 0.96900 * b2 + white * 0.1538520;
  b3 = 0.86650 * b3 + white * 0.3104856;
  b4 = 0.55000 * b4 + white * 0.5329522;
  b5 = -0.7616 * b5 - white * 0.0168980;
  pink = b0 + b1 + b2 + b3 + b4 + b5 + b6 + white * 0.5362;
  b6 = white * 0.115926;


An 'economy' version with accuracy of +/-0.5dB is also available.

  b0 = 0.99765 * b0 + white * 0.0990460;
  b1 = 0.96300 * b1 + white * 0.2965164;
  b2 = 0.57000 * b2 + white * 1.0526913;
  pink = b0 + b1 + b2 + white * 0.1848;



---
paul.kellett@maxim.abel.co.uk
http://www.abel.co.uk/~maxim/
```

Polyphase Filters (click this to go back to the index)

Type : polyphase filters, used for up and down-sampling

References : C++ source code by Dave from Muon Software

Linked file : BandLimit.cpp (this linked file is included below)

Linked file : BandLimit.h (this linked file is included below)

Linked files

```
CAllPassFilter::CAllPassFilter(const double coefficient)
{
 a=coefficient;

 x0=0.0;
 x1=0.0;
 x2=0.0;

 y0=0.0;
 y1=0.0;
 y2=0.0;

};


CAllPassFilter::~CAllPassFilter()
{
};


double CAllPassFilter::process(const double input)
{
 //shuffle inputs
 x2=x1;
 x1=x0;
 x0=input;

 //shuffle outputs
 y2=y1;
 y1=y0;

 //allpass filter 1
 const double output=x2+((input-y2)*a);

 y0=output;

 return output;
};


CAllPassFilterCascade::CAllPassFilterCascade(const double* coefficient, const int N)
{
 allpassfilter=new CAllPassFilter*[N];

 for (int i=0;i<N;i++)
 {
  allpassfilter[i]=new CAllPassFilter(coefficient[i]);
 }

 numfilters=N;
};


CAllPassFilterCascade::~CAllPassFilterCascade()
{
 delete[] allpassfilter;
};


double CAllPassFilterCascade::process(const double input)
{
 double output=input;

 int i=0;

 do
 {
  output=allpassfilter[i]->process(output);
  i++;
```

```
 } while (i<numfilters);

 return output;
};


CHalfBandFilter::CHalfBandFilter(const int order, const bool steep)
{
 if (steep==true)
 {
  if (order==12) //rejection=104dB, transition band=0.01
  {
   double a_coefficients[6]=
   {0.036681502163648017
   ,0.2746317593794541
   ,0.56109896978791948
   ,0.769741833862266
   ,0.8922608180038789
   ,0.962094548378084
   };

   double b_coefficients[6]=
   {0.13654762463195771
   ,0.42313861743656667
   ,0.6775400499741616
   ,0.839889624849638
   ,0.9315419599631839
   ,0.9878163707328971
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,6);
   filter_b=new CAllPassFilterCascade(b_coefficients,6);
  }
  else if (order==10) //rejection=86dB, transition band=0.01
  {
   double a_coefficients[5]=
   {0.051457617441190984
   ,0.35978656070567017
   ,0.6725475931034693
   ,0.8590884928249939
   ,0.9540209867860787
   };

   double b_coefficients[5]=
   {0.18621906251989334
   ,0.529951372847964
   ,0.7810257527489514
   ,0.9141815687605308
   ,0.985475023014907
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,5);
   filter_b=new CAllPassFilterCascade(b_coefficients,5);
  }
  else if (order==8) //rejection=69dB, transition band=0.01
  {
   double a_coefficients[4]=
   {0.07711507983241622
   ,0.4820706250610472
   ,0.7968204713315797
   ,0.9412514277740471
   };

   double b_coefficients[4]=
   {0.2659685265210946
   ,0.6651041532634957
   ,0.8841015085506159
   ,0.9820054141886075
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,4);
   filter_b=new CAllPassFilterCascade(b_coefficients,4);
  }
  else if (order==6) //rejection=51dB, transition band=0.01
  {
   double a_coefficients[3]=
   {0.1271414136264853
   ,0.6528245886369117
   ,0.9176942834328115
   };

   double b_coefficients[3]=
   {0.40056789819445626
```

```
   ,0.8204163891923343
   ,0.9763114515836773
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,3);
   filter_b=new CAllPassFilterCascade(b_coefficients,3);
  }
  else if (order==4) //rejection=53dB,transition band=0.05
  {
   double a_coefficients[2]=
   {0.12073211751675449
   ,0.6632020224193995
   };

   double b_coefficients[2]=
   {0.3903621872345006
   ,0.890786832653497
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,2);
   filter_b=new CAllPassFilterCascade(b_coefficients,2);
  }

  else //order=2, rejection=36dB, transition band=0.1
  {
   double a_coefficients=0.23647102099689224;
   double b_coefficients=0.7145421497126001;

   filter_a=new CAllPassFilterCascade(&a_coefficients,1);
   filter_b=new CAllPassFilterCascade(&b_coefficients,1);
  }
 }
 else //softer slopes, more attenuation and less stopband ripple
 {
  if (order==12) //rejection=150dB, transition band=0.05
  {
   double a_coefficients[6]=
   {0.01677466677723562
   ,0.13902148819717805
   ,0.3325011117394731
   ,0.53766105314488
   ,0.7214184024215805
   ,0.8821858402078155
   };

   double b_coefficients[6]=
   {0.06501319274445962
   ,0.23094129990840923
   ,0.4364942348420355
   ,0.06329609551399348
   ,0.80378086794111226
   ,0.9599687404800694
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,6);
   filter_b=new CAllPassFilterCascade(b_coefficients,6);
  }
  else if (order==10) //rejection=133dB, transition band=0.05
  {
   double a_coefficients[5]=
   {0.02366831419883467
   ,0.18989476227180174
   ,0.43157318062118555
   ,0.6632020224193995
   ,0.860015542499582
   };

   double b_coefficients[5]=
   {0.09056555904993387
   ,0.3078575723749043
   ,0.5516782402507934
   ,0.7652146863779808
   ,0.95247728378667541
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,5);
   filter_b=new CAllPassFilterCascade(b_coefficients,5);
  }
  else if (order==8) //rejection=106dB, transition band=0.05
  {
   double a_coefficients[4]=
   {0.03583278843106211
   ,0.2720401433964576
```

```
   ,0.5720571972357003
   ,0.827124761997324
   };

   double b_coefficients[4]=
   {0.1340901419430669
   ,0.4243248712718685
   ,0.7062921421386394
   ,0.9415030941737551
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,4);
   filter_b=new CAllPassFilterCascade(b_coefficients,4);
  }
  else if (order==6) //rejection=80dB, transition band=0.05
  {
   double a_coefficients[3]=
   {0.06029739095712437
   ,0.4125907203610563
   ,0.7727156537429234
   };

   double b_coefficients[3]=
   {0.21597144456092948
   ,0.6043586264658363
   ,0.9238861386532906
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,3);
   filter_b=new CAllPassFilterCascade(b_coefficients,3);
  }
  else if (order==4) //rejection=70dB,transition band=0.1
  {
   double a_coefficients[2]=
   {0.07986642623635751
   ,0.5453536510711322
   };

   double b_coefficients[2]=
   {0.28382934487410993
   ,0.8344118914807379
   };

   filter_a=new CAllPassFilterCascade(a_coefficients,2);
   filter_b=new CAllPassFilterCascade(b_coefficients,2);
  }

  else //order=2, rejection=36dB, transition band=0.1
  {
   double a_coefficients=0.23647102099689224;
   double b_coefficients=0.7145421497126001;

   filter_a=new CAllPassFilterCascade(&a_coefficients,1);
   filter_b=new CAllPassFilterCascade(&b_coefficients,1);
  }
 }

 oldout=0.0;
};


CHalfBandFilter::~CHalfBandFilter()
{
 delete filter_a;
 delete filter_b;
};


double CHalfBandFilter::process(const double input)
{
 const double output=(filter_a->process(input)+oldout)*0.5;
 oldout=filter_b->process(input);

 return output;
}
class CAllPassFilter
{
public:

 CAllPassFilter(const double coefficient);
 ~CAllPassFilter();
 double process(double input);
```

```
private:
 double a;

 double x0;
 double x1;
 double x2;

 double y0;
 double y1;
 double y2;
};


class CAllPassFilterCascade
{
public:
 CAllPassFilterCascade(const double* coefficients, int N);
 ~CAllPassFilterCascade();

 double process(double input);

private:
 CAllPassFilter** allpassfilter;
 int numfilters;
};


class CHalfBandFilter
{
public:
 CHalfBandFilter(const int order, const bool steep);
 ~CHalfBandFilter();

 double process(const double input);

private:
 CAllPassFilterCascade* filter_a;
 CAllPassFilterCascade* filter_b;
 double oldout;
};
```

Prewarping (click this to go back to the index)

Type : explanation

References : Posted by robert bristow-johnson (better known as "rbj")

Notes :
prewarping is simply recognizing the warping that the BLT introduces.
to determine frequency response, we evaluate the digital H(z) at
z=exp(j*w*T) and we evaluate the analog Ha(s) at s=j*W . the following
will confirm the jw to unit circle mapping and will show exactly what the
mapping is (this is the same stuff in the textbooks):

the BLT says: s = (2/T) * (z-1)/(z+1)

substituting: s = j*W = (2/T) * (exp(j*w*T) - 1) / (exp(j*w*T) + 1)

j*W = (2/T) * (exp(j*w*T/2) - exp(-j*w*T/2)) / (exp(j*w*T/2) + exp(-j*w*T/2))

= (2/T) * (j*2*sin(w*T/2)) / (2*cos(w*T/2))

= j * (2/T) * tan(w*T/2)

or

analog W = (2/T) * tan(w*T/2)

so when the real input frequency is w, the digital filter will behave with
the same amplitude gain and phase shift as the analog filter will have at a
hypothetical frequency of W. as w*T approaches pi (Nyquist) the digital
filter behaves as the analog filter does as W -> inf. for each degree of
freedom that you have in your design equations, you can adjust the analog
design frequency to be just right so that when the deterministic BLT
warping does its thing, the resultant warped frequency comes out just
right. for a simple LPF, you have only one degree of freedom, the cutoff
frequency. you can precompensate it so that the true cutoff comes out
right but that is it, above the cutoff, you will see that the LPF dives
down to -inf dB faster than an equivalent analog at the same frequencies.

RBJ-Audio-EQ-Cookbook (click this to go back to the index)

Type : Biquads for all puposes!

References : Robert Bristow-Johnson (a.k.a. RBJ)

Linked file : http://www.harmony-central.com/Computer/Programming/Audio-EQ-Cookbook.txt

Linked file : http://www.harmony-central.com/Effects/Articles/EQ_Coefficients/EQ-Coefficients.pdf

Notes :
(see linkfile)
A superb collection of filters used in a lot of (commercial) plugins and effects.
There's also a very interesting paper linked about biquad EQ filters.

Resonant filter (click this to go back to the index)

References : Posted by Paul Kellett

Notes :
This filter consists of two first order low-pass filters in series, with some of the difference between the two filter outputs fed back to give a resonant peak.

You can use more filter stages for a steeper cutoff but the stability criteria get more complicated if the extra stages are within the feedback loop.

Code :
```
//set feedback amount given f and q between 0 and 1
fb = q + q/(1.0 - f);

//for each sample...
buf0 = buf0 + f * (in - buf0 + fb * (buf0 - buf1));
buf1 = buf1 + f * (buf0 - buf1);
out = buf1;
```

Resonant IIR lowpass (12dB/oct) (click this to go back to the index)

Type : Resonant IIR lowpass (12dB/oct)

References : Posted by Olli Niemitalo

Notes :
Hard to calculate coefficients, easy to process algorithm

Code :
```
resofreq = pole frequency
amp = magnitude at pole frequency (approx)

double pi = 3.141592654;

/* Parameters. Change these! */
double resofreq = 5000;
double amp = 1.0;

DOUBLEWORD streamofs;
double w = 2.0*pi*resofreq/samplerate; // Pole angle
double q = 1.0-w/(2.0*(amp+0.5/(1.0+w))+w-2.0); // Pole magnitude
double r = q*q;
double c = r+1.0-2.0*cos(w)*q;
double vibrapos = 0;
double vibraspeed = 0;

/* Main loop */
for (streamofs = 0; streamofs < streamsize; streamofs++) {

  /* Accelerate vibra by signal-vibra, multiplied by lowpasscutoff */
  vibraspeed += (fromstream[streamofs] - vibrapos) * c;

  /* Add velocity to vibra's position */
  vibrapos += vibraspeed;

  /* Attenuate/amplify vibra's velocity by resonance */
  vibraspeed *= r;

  /* Check clipping */
  temp = vibrapos;
  if (temp > 32767) {
    temp = 32767;
  } else if (temp < -32768) temp = -32768;

  /* Store new value */
  tostream[streamofs] = temp;
}
```

Comments
from : raucous@attbi.com
comment : This looks similar to the low-pass filter I used in FilterKing (http://home.attbi.com/~spaztek4/) Can you cruft up a high-pass example for me?

Thanks,
__e

Resonant low pass filter (click this to go back to the index)

Type : 24dB lowpass

References : Posted by "Zxform"

Linked file : filters004.txt (this linked file is included below)

Linked files

```
// ----------------- file filterIIR00.c begin -----------------
/*
Resonant low pass filter source code.
By baltrax@hotmail.com (Zxform)
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/**************************************************************************

FILTER.C - Source code for filter functions

    iir_filter         IIR filter floats sample by sample (real time)

*************************************************************************/

/* FILTER INFORMATION STRUCTURE FOR FILTER ROUTINES */

typedef struct {
    unsigned int length;       /* size of filter */
    float *history;            /* pointer to history in filter */
    float *coef;               /* pointer to coefficients of filter */
} FILTER;

#define FILTER_SECTIONS   2   /* 2 filter sections for 24 db/oct filter */

typedef struct {
        double a0, a1, a2;       /* numerator coefficients */
        double b0, b1, b2;       /* denominator coefficients */
} BIQUAD;

BIQUAD ProtoCoef[FILTER_SECTIONS];      /* Filter prototype coefficients,
                                                     1 for each filter section
*/

void szxform(
    double *a0, double *a1, double *a2,     /* numerator coefficients */
    double *b0, double *b1, double *b2,   /* denominator coefficients */
    double fc,           /* Filter cutoff frequency */
    double fs,           /* sampling rate */
    double *k,           /* overall gain factor */
    float *coef);         /* pointer to 4 iir coefficients */

/*
 * --------------------------------------------------------------------
 *
 * iir_filter - Perform IIR filtering sample by sample on floats
 *
 * Implements cascaded direct form II second order sections.
 * Requires FILTER structure for history and coefficients.
 * The length in the filter structure specifies the number of sections.
 * The size of the history array is 2*iir->length.
 * The size of the coefficient array is 4*iir->length + 1 because
 * the first coefficient is the overall scale factor for the filter.
 * Returns one output sample for each input sample.  Allocates history
 * array if not previously allocated.
 *
 * float iir_filter(float input,FILTER *iir)
 *
 *     float input        new float input sample
 *     FILTER *iir        pointer to FILTER structure
 *
 * Returns float value giving the current output.
 *
 * Allocation errors cause an error message and a call to exit.
 * --------------------------------------------------------------------
 */
float iir_filter(input,iir)
    float input;        /* new input sample */
    FILTER *iir;        /* pointer to FILTER structure */
```

```
{
    unsigned int i;
    float *hist1_ptr,*hist2_ptr,*coef_ptr;
    float output,new_hist,history1,history2;

/* allocate history array if different size than last call */

    if(!iir->history) {
        iir->history = (float *) calloc(2*iir->length,sizeof(float));
        if(!iir->history) {
            printf("\nUnable to allocate history array in iir_filter\n");
            exit(1);
        }
    }

    coef_ptr = iir->coef;                /* coefficient pointer */

    hist1_ptr = iir->history;            /* first history */
    hist2_ptr = hist1_ptr + 1;           /* next history */

        /* 1st number of coefficients array is overall input scale factor,
         * or filter gain */
    output = input * (*coef_ptr++);

    for (i = 0 ; i < iir->length; i++)
        {
        history1 = *hist1_ptr;           /* history values */
        history2 = *hist2_ptr;

        output = output - history1 * (*coef_ptr++);
        new_hist = output - history2 * (*coef_ptr++);    /* poles */

        output = new_hist + history1 * (*coef_ptr++);
        output = output + history2 * (*coef_ptr++);      /* zeros */

        *hist2_ptr++ = *hist1_ptr;
        *hist1_ptr++ = new_hist;
        hist1_ptr++;
        hist2_ptr++;
    }

    return(output);
}


/*
 * ----------------------------------------------------------------------
 *
 * main()
 *
 * Example main function to show how to update filter coefficients.
 * We create a 4th order filter (24 db/oct roloff), consisting
 * of two second order sections.
 * ----------------------------------------------------------------------
 */
int main()
{
        FILTER   iir;
        float    *coef;
        double   fs, fc;     /* Sampling frequency, cutoff frequency */
        double   Q;     /* Resonance > 1.0 < 1000 */
        unsigned nInd;
        double   a0, a1, a2, b0, b1, b2;
        double   k;           /* overall gain factor */

/*
 * Setup filter s-domain coefficients
 */
                 /* Section 1 */
        ProtoCoef[0].a0 = 1.0;
        ProtoCoef[0].a1 = 0;
        ProtoCoef[0].a2 = 0;
        ProtoCoef[0].b0 = 1.0;
        ProtoCoef[0].b1 = 0.765367;
        ProtoCoef[0].b2 = 1.0;

                 /* Section 2 */
        ProtoCoef[1].a0 = 1.0;
        ProtoCoef[1].a1 = 0;
        ProtoCoef[1].a2 = 0;
        ProtoCoef[1].b0 = 1.0;
        ProtoCoef[1].b1 = 1.847759;
        ProtoCoef[1].b2 = 1.0;
```

```
        iir.length = FILTER_SECTIONS;         /* Number of filter sections */

/*
 * Allocate array of z-domain coefficients for each filter section
 * plus filter gain variable
 */
        iir.coef = (float *) calloc(4 * iir.length + 1, sizeof(float));
        if (!iir.coef)
        {
                 printf("Unable to allocate coef array, exiting\n");
                 exit(1);
        }

        k = 1.0;          /* Set overall filter gain */
        coef = iir.coef + 1;     /* Skip k, or gain */

        Q = 1;                         /* Resonance */
        fc = 5000;                  /* Filter cutoff (Hz) */
        fs = 44100;                      /* Sampling frequency (Hz) */

/*
 * Compute z-domain coefficients for each biquad section
 * for new Cutoff Frequency and Resonance
 */
        for (nInd = 0; nInd < iir.length; nInd++)
        {
                 a0 = ProtoCoef[nInd].a0;
                 a1 = ProtoCoef[nInd].a1;
                 a2 = ProtoCoef[nInd].a2;

                 b0 = ProtoCoef[nInd].b0;
                 b1 = ProtoCoef[nInd].b1 / Q;      /* Divide by resonance or Q
*/
                 b2 = ProtoCoef[nInd].b2;
                 szxform(&a0, &a1, &a2, &b0, &b1, &b2, fc, fs, &k, coef);
                 coef += 4;                       /* Point to next filter
section */
        }

        /* Update overall filter gain in coef array */
        iir.coef[0] = k;

        /* Display filter coefficients */
        for (nInd = 0; nInd < (iir.length * 4 + 1); nInd++)
                 printf("C[%d] = %15.10f\n", nInd, iir.coef[nInd]);
/*
 * To process audio samples, call function iir_filter()
 * for each audio sample
 */
        return (0);
}




// ----------------- file filterIIR00.c end -----------------
>
```

Reposting bilinear.c just in case the other one was not the latest version.

```
// ----------------- file bilinear.c begin -----------------
/*
 * ----------------------------------------------------------
 *      bilinear.c
 *
 *      Perform bilinear transformation on s-domain coefficients
 *      of 2nd order biquad section.
 *      First design an analog filter and use s-domain coefficients
 *      as input to szxform() to convert them to z-domain.
 *
 * Here's the butterworth polinomials for 2nd, 4th and 6th order sections.
 *      When we construct a 24 db/oct filter, we take to 2nd order
 *      sections and compute the coefficients separately for each section.
 *
 *      n       Polinomials
 * --------------------------------------------------------------------
 *      2       s^2 + 1.4142s +1
 *      4       (s^2 + 0.765367s + 1) (s^2 + 1.847759s + 1)
 *      6       (s^2 + 0.5176387s + 1) (s^2 + 1.414214 + 1) (s^2 + 1.931852s +
1)
 *
 *      Where n is a filter order.
```

```
 *      For n=4, or two second order sections, we have following equasions for
each
 *      2nd order stage:
 *
 *      (1 / (s^2 + (1/Q) * 0.765367s + 1)) * (1 / (s^2 + (1/Q) * 1.847759s +
1))
 *
 *      Where Q is filter quality factor in the range of
 *      1 to 1000. The overall filter Q is a product of all
 *      2nd order stages. For example, the 6th order filter
 *      (3 stages, or biquads) with individual Q of 2 will
 *      have filter Q = 2 * 2 * 2 = 8.
 *
 *      The nominator part is just 1.
 *      The denominator coefficients for stage 1 of filter are:
 *      b2 = 1; b1 = 0.765367; b0 = 1;
 *      numerator is
 *      a2 = 0; a1 = 0; a0 = 1;
 *
 *      The denominator coefficients for stage 1 of filter are:
 *      b2 = 1; b1 = 1.847759; b0 = 1;
 *      numerator is
 *      a2 = 0; a1 = 0; a0 = 1;
 *
 *      These coefficients are used directly by the szxform()
 *      and bilinear() functions. For all stages the numerator
 *      is the same and the only thing that is different between
 *      different stages is 1st order coefficient. The rest of
 *      coefficients are the same for any stage and equal to 1.
 *
 *      Any filter could be constructed using this approach.
 *
 *      References:
 *             Van Valkenburg, "Analog Filter Design"
 *             Oxford University Press 1982
 *             ISBN 0-19-510734-9
 *
 *             C Language Algorithms for Digital Signal Processing
 *             Paul Embree, Bruce Kimble
 *             Prentice Hall, 1991
 *             ISBN 0-13-133406-9
 *
 *             Digital Filter Designer's Handbook
 *             With C++ Algorithms
 *             Britton Rorabaugh
 *             McGraw Hill, 1997
 *             ISBN 0-07-053806-9
 * ----------------------------------------------------------
 */

#include <math.h>

void prewarp(double *a0, double *a1, double *a2, double fc, double fs);
void bilinear(
    double a0, double a1, double a2,    /* numerator coefficients */
    double b0, double b1, double b2,    /* denominator coefficients */
    double *k,                                   /* overall gain factor */
    double fs,                                   /* sampling rate */
    float *coef);                         /* pointer to 4 iir coefficients */


/*
 * ----------------------------------------------------------
 *      Pre-warp the coefficients of a numerator or denominator.
 *      Note that a0 is assumed to be 1, so there is no wrapping
 *      of it.
 * ----------------------------------------------------------
 */
void prewarp(
    double *a0, double *a1, double *a2,
    double fc, double fs)
{
    double wp, pi;

    pi = 4.0 * atan(1.0);
    wp = 2.0 * fs * tan(pi * fc / fs);

    *a2 = (*a2) / (wp * wp);
    *a1 = (*a1) / wp;
}


/*
```

```
 * ----------------------------------------------------------
 * bilinear()
 *
 * Transform the numerator and denominator coefficients
 * of s-domain biquad section into corresponding
 * z-domain coefficients.
 *
 *      Store the 4 IIR coefficients in array pointed by coef
 *      in following order:
 *             beta1, beta2    (denominator)
 *             alpha1, alpha2  (numerator)
 *
 * Arguments:
 *             a0-a2   - s-domain numerator coefficients
 *             b0-b2   - s-domain denominator coefficients
 *             k               - filter gain factor. initially set to 1
 *                                and modified by each biquad section in such
 *                                a way, as to make it the coefficient by
 *                                which to multiply the overall filter gain
 *                                in order to achieve a desired overall filter
gain,
 *                                specified in initial value of k.
 *             fs             - sampling rate (Hz)
 *             coef    - array of z-domain coefficients to be filled in.
 *
 * Return:
 *             On return, set coef z-domain coefficients
 * ----------------------------------------------------------
 */
void bilinear(
    double a0, double a1, double a2,    /* numerator coefficients */
    double b0, double b1, double b2,    /* denominator coefficients */
    double *k,           /* overall gain factor */
    double fs,           /* sampling rate */
    float *coef         /* pointer to 4 iir coefficients */
)
{
    double ad, bd;

                 /* alpha (Numerator in s-domain) */
    ad = 4. * a2 * fs * fs + 2. * a1 * fs + a0;
                 /* beta (Denominator in s-domain) */
    bd = 4. * b2 * fs * fs + 2. * b1* fs + b0;

                 /* update gain constant for this section */
    *k *= ad/bd;

                 /* Denominator */
    *coef++ = (2. * b0 - 8. * b2 * fs * fs)
                           / bd;         /* beta1 */
    *coef++ = (4. * b2 * fs * fs - 2. * b1 * fs + b0)
                           / bd; /* beta2 */

                 /* Nominator */
    *coef++ = (2. * a0 - 8. * a2 * fs * fs)
                           / ad;         /* alpha1 */
    *coef = (4. * a2 * fs * fs - 2. * a1 * fs + a0)
                           / ad;   /* alpha2 */
}


/*
 * ----------------------------------------------------------
 * Transform from s to z domain using bilinear transform
 * with prewarp.
 *
 * Arguments:
 *      For argument description look at bilinear()
 *
 *      coef - pointer to array of floating point coefficients,
 *                     corresponding to output of bilinear transofrm
 *                     (z domain).
 *
 * Note: frequencies are in Hz.
 * ----------------------------------------------------------
 */
void szxform(
    double *a0, double *a1, double *a2, /* numerator coefficients */
    double *b0, double *b1, double *b2, /* denominator coefficients */
    double fc,         /* Filter cutoff frequency */
    double fs,         /* sampling rate */
    double *k,         /* overall gain factor */
    float *coef)         /* pointer to 4 iir coefficients */
```

```
{
                 /* Calculate a1 and a2 and overwrite the original values */
        prewarp(a0, a1, a2, fc, fs);
        prewarp(b0, b1, b2, fc, fs);
        bilinear(*a0, *a1, *a2, *b0, *b1, *b2, k, fs, coef);
}


// ----------------- file bilinear.c end -----------------

And here is how it all works.

// ----------------- file filter.txt begin -----------------
How to construct a kewl low pass resonant filter?

Lets assume we want to create a filter for analog synth.
The filter rolloff is 24 db/oct, which corresponds to 4th
order filter. Filter of first order is equivalent to RC circuit
and has max rolloff of 6 db/oct.

We will use classical Butterworth IIR filter design, as it
exactly corresponds to our requirements.

A common practice is to chain several 2nd order sections,
or biquads, as they commonly called, in order to achive a higher
order filter. Each 2nd order section is a 2nd order filter, which
has 12 db/oct roloff. So, we need 2 of those sections in series.

To compute those sections, we use standard Butterworth polinomials,
or so called s-domain representation and convert it into z-domain,
or digital domain. The reason we need to do this is because
the filter theory exists for analog filters for a long time
and there exist no theory of working in digital domain directly.
So the common practice is to take standard analog filter design
and use so called bilinear transform to convert the butterworth
equasion coefficients into z-domain.

Once we compute the z-domain coefficients, we can use them in
a very simple transfer function, such as iir_filter() in our
C source code, in order to perform the filtering function.
The filter itself is the simpliest thing in the world.
The most complicated thing is computing the coefficients
for z-domain.

Ok, lets look at butterworth polynomials, arranged as a series
of 2nd order sections:

 * Note: n is filter order.
 *
 *      n       Polynomials
 *      --------------------------------------------------------------------
 *      2       s^2 + 1.4142s +1
 *      4       (s^2 + 0.765367s + 1) * (s^2 + 1.847759s + 1)
 *      6       (s^2 + 0.5176387s + 1) * (s^2 + 1.414214 + 1) * (s^2 +
1.931852s + 1)
 *
 * For n=4 we have following equasion for the filter transfer function:
 *
 *                     1                              1
 * T(s) = --------------------------- * ----------------------------
 *        s^2 + (1/Q) * 0.765367s + 1   s^2 + (1/Q) * 1.847759s + 1
 *

The filter consists of two 2nd order secions since highest s power is 2.
Now we can take the coefficients, or the numbers by which s is multiplied
and plug them into a standard formula to be used by bilinear transform.

Our standard form for each 2nd order secion is:

       a2 * s^2 + a1 * s + a0
H(s) = ----------------------
       b2 * s^2 + b1 * s + b0

Note that butterworth nominator is 1 for all filter sections,
which means s^2 = 0 and s^1 = 0

Lets convert standard butterworth polinomials into this form:

       0 + 0 + 1                  0 + 0 + 1
--------------------------- * --------------------------
1 + ((1/Q) * 0.765367) + 1   1 + ((1/Q) * 1.847759) + 1

Section 1:
```

```
a2 = 0; a1 = 0; a0 = 1;
b2 = 1; b1 = 0.5176387; b0 = 1;

Section 2:
a2 = 0; a1 = 0; a0 = 1;
b2 = 1; b1 = 1.847759; b0 = 1;

That Q is filter quality factor or resonance, in the range of
1 to 1000. The overall filter Q is a product of all 2nd order stages.
For example, the 6th order filter (3 stages, or biquads)
with individual Q of 2 will have filter Q = 2 * 2 * 2 = 8.

These a and b coefficients are used directly by the szxform()
and bilinear() functions.

The transfer function for z-domain is:

       1 + alpha1 * z^(-1) + alpha2 * z^(-2)
H(z) = -------------------------------------
       1 + beta1 * z^(-1) + beta2 * z^(-2)

When you need to change the filter frequency cutoff or resonance,
or Q, you call the szxform() function with proper a and b
coefficients and the new filter cutoff frequency or resonance.
You also need to supply the sampling rate and filter gain you want
to achive. For our purposes the gain = 1.

We call szxform() function 2 times becase we have 2 filter sections.
Each call provides different coefficients.

The gain argument to szxform() is a pointer to desired filter
gain variable.

double k = 1.0;      /* overall gain factor */

Upon return from each call, the k argument will be set to a value,
by which to multiply our actual signal in order for the gain
to be one. On second call to szxform() we provide k that was
changed by the previous section. During actual audio filtering
function iir_filter() will use this k

Summary:

Our filter is pretty close to ideal in terms of all relevant
parameters and filter stability even with extremely large values
of resonance. This filter design has been verified under all
variations of parameters and it all appears to work as advertized.

Good luck with it.
If you ever make a directX wrapper for it, post it to comp.dsp.


 *
 * ----------------------------------------------------------
 *References:
 *Van Valkenburg, "Analog Filter Design"
 *Oxford University Press 1982
 *ISBN 0-19-510734-9
 *
 *C Language Algorithms for Digital Signal Processing
 *Paul Embree, Bruce Kimble
 *Prentice Hall, 1991
 *ISBN 0-13-133406-9
 *
 *Digital Filter Designer's Handbook
 *With C++ Algorithms
 *Britton Rorabaugh
 *McGraw Hill, 1997
 *ISBN 0-07-053806-9
 * ----------------------------------------------------------
```

State variable (click this to go back to the index)

Type : 12db resonant low, high or bandpass

References : Effect Deisgn Part 1, Jon Dattorro, J. Audio Eng. Soc., Vol 45, No. 9, 1997 September

Notes :
Digital approximation of Chamberlin two-pole low pass. Easy to calculate coefficients, easy to process algorithm.

Code :
```
cutoff = cutoff freq in Hz
fs = sampling frequency //(e.g. 44100Hz)
f = 2 sin (pi * cutoff / fs) //[approximately]
q = resonance/bandwidth [0 < q <= 1]  most res: q=1, less: q=0
low = lowpass output
high = highpass output
band = bandpass output
notch = notch output

scale = q

low=high=band=0;

//--beginloop
low = low + f * band;
high = scale * input - low - q*band;
band = f * high + band;
notch = high + low;
//--endloop
```

State Variable Filter (Chamberlin version) (click this to go back to the index)

References : Hal Chamberlin, "Musical Applications of Microprocessors," 2nd Ed, Hayden Book Company 1985. pp 490-492.

Code :
```
//Input/Output
 I - input sample
 L - lowpass output sample
 B - bandpass output sample
 H - highpass output sample
 N - notch output sample
 F1 - Frequency control parameter
 Q1 - Q control parameter
 D1 - delay associated with bandpass output
 D2 - delay associated with low-pass output

// parameters:
 Q1 = 1/Q
 // where Q1 goes from 2 to 0, ie Q goes from .5 to infinity

 // simple frequency tuning with error towards nyquist
 // F is the filter's center frequency, and Fs is the sampling rate
 F1 = 2*pi*F/Fs

 // ideal tuning:
 F1 = 2 * sin( pi * F / Fs )

// algorithm
 // loop
 L = D2 + F1 * D1
 H = I - L - Q1*D1
 B = F1 * H + D1
 N = H + L

 // store delays
 D1 = B
 D2 = L

 // outputs
 L,H,B,N
```

State Variable Filter (Double Sampled, Stable) (click this to go back to the index)

Type : 2 Pole Low, High, Band, Notch and Peaking

References : Posted by Andrew Simper

Notes :
Thanks to Laurent de Soras for the stability limit
and Steffan Diedrichsen for the correct notch output.

Code :
```
input  = input buffer;
output = output buffer;
fs     = sampling frequency;
fc     = cutoff frequency normally something like:
         440.0*pow(2.0, (midi_note - 69.0)/12.0);
res    = resonance 0 to 1;
drive  = internal distortion 0 to 0.1
freq   = 2.0*sin(PI*MIN(0.25, fc/(fs*2)));  // the fs*2 is because it's double sampled
damp   = MIN(2.0*(1.0 - pow(res, 0.25)), MIN(2.0, 2.0/freq - freq*0.5));
notch  = notch output
low    = low pass output
high   = high pass output
band   = band pass output
peak   = peaking output = low - high
--
double sampled svf loop:
for (i=0; i<numSamples; i++)
{
  in    = input[i];
  notch = in - damp*band;
  low   = low + freq*band;
  high  = notch - low;
  band  = freq*high + band - drive*band*band*band;
  out   = 0.5*(notch or low or high or band or peak);
  notch = in - damp*band;
  low   = low + freq*band;
  high  = notch - low;
  band  = freq*high + band - drive*band*band*band;
  out  += 0.5*(same out as above);
  output[i] = out;
}
```

Stilson's Moog filter code (click this to go back to the index)

Type : 4-pole LP, with fruity BP/HP

References : Posted by DFL

Notes :
Mind your p's and Q's...

This code was borrowed from Tim Stilson, and rewritten by me into a pd extern (moog~) available here:
http://www-ccrma.stanford.edu/~dfl/pd/index.htm

I ripped out the essential code and pasted it here...

Code :
```
WARNING: messy code follows ;)

// table to fixup Q in order to remain constant for various pole frequencies, from Tim Stilson's code @ CCRMA
(also in CLM distribution)

static float gaintable[199] = { 0.999969, 0.990082, 0.980347, 0.970764, 0.961304, 0.951996, 0.94281, 0.933777,
0.924866, 0.916077, 0.90741, 0.898865, 0.89044
2, 0.882141 , 0.873962, 0.865906, 0.857941, 0.850067, 0.842346, 0.834686, 0.827148, 0.819733, 0.812378,
0.805145, 0.798004, 0.790955, 0.783997, 0.77713, 0.77
0355, 0.763672, 0.75708 , 0.75058, 0.744141, 0.737793, 0.731537, 0.725342, 0.719238, 0.713196, 0.707245,
0.701355, 0.695557, 0.689819, 0.684174, 0.678558, 0.
673035, 0.667572, 0.66217, 0.65686, 0.651581, 0.646393, 0.641235, 0.636169, 0.631134, 0.62619, 0.621277,
0.616425, 0.611633, 0.606903, 0.602234, 0.597626, 0.
593048, 0.588531, 0.584045, 0.579651, 0.575287 , 0.570953, 0.566681, 0.562469, 0.558289, 0.554169, 0.550079,
0.546051, 0.542053, 0.538116, 0.53421, 0.530334,
 0.52652, 0.522736, 0.518982, 0.515289, 0.511627, 0.507996 , 0.504425, 0.500885, 0.497375, 0.493896, 0.490448,
0.487061, 0.483704, 0.480377, 0.477081, 0.4738
16, 0.470581, 0.467377, 0.464203, 0.46109, 0.457977, 0.454926, 0.451874, 0.448883, 0.445892, 0.442932,
0.440033, 0.437134, 0.434265, 0.431427, 0.428619, 0.42
5842, 0.423096, 0.42038, 0.417664, 0.415009, 0.412354, 0.409729, 0.407135, 0.404572, 0.402008, 0.399506,
0.397003, 0.394501, 0.392059, 0.389618, 0.387207, 0.
384827, 0.382477, 0.380127, 0.377808, 0.375488, 0.37323, 0.370972, 0.368713, 0.366516, 0.364319, 0.362122,
0.359985, 0.357849, 0.355713, 0.353607, 0.351532,
0.349457, 0.347412, 0.345398, 0.343384, 0.34137, 0.339417, 0.337463, 0.33551, 0.333588, 0.331665, 0.329773,
0.327911, 0.32605, 0.324188, 0.322357, 0.320557,
0.318756, 0.316986, 0.315216, 0.313446, 0.311707, 0.309998, 0.308289, 0.30658, 0.304901, 0.303223, 0.301575,
0.299927, 0.298309, 0.296692, 0.295074, 0.293488
, 0.291931, 0.290375, 0.288818, 0.287262, 0.285736, 0.284241, 0.282715, 0.28125, 0.279755, 0.27829, 0.276825,
0.275391, 0.273956, 0.272552, 0.271118, 0.26974
5, 0.268341, 0.266968, 0.265594, 0.264252, 0.262909, 0.261566, 0.260223, 0.258911, 0.257599, 0.256317,
0.255035, 0.25375 };

static inline float saturate( float input ) { //clamp without branching
#define _limit 0.95
  float x1 = fabsf( input + _limit );
  float x2 = fabsf( input - _limit );
  return 0.5 * (x1 - x2);
}

static inline float crossfade( float amount, float a, float b ) {
  return (1-amount)*a + amount*b;
}

//code for setting Q
        float ix, ixfrac;
        int ixint;
        ix = x->p * 99;
        ixint = floor( ix );
        ixfrac = ix - ixint;
     Q = resonance * crossfade( ixfrac, gaintable[ ixint + 99 ], gaintable[ ixint + 100 ] );

//code for setting pole coefficient based on frequency
    float fc = 2 * frequency / x->srate;
    float x2 = fc*fc;
    float x3 = fc*x2;
    p = -0.69346 * x3 - 0.59515 * x2 + 3.2937 * fc - 1.0072; //cubic fit by DFL, not 100% accurate but better
than nothing...
}


process loop:
  float state[4], output; //should be global scope / preserved between calls
  int i,pole;
  float temp, input;

  for ( i=0; i < numSamples; i++ ) {
          input = *(in++);
          output = 0.25 * ( input - output ); //negative feedback

          for( pole = 0; pole < 4; pole++) {
                  temp = state[pole];
                  output = saturate( output + p * (output - temp));
                  state[pole] = output;
                  output = saturate( output + temp );
          }
```

```
        lowpass = output;
        highpass = input - output;
        bandpass = 3 * x->state[2] - x->lowpass; //got this one from paul kellet
        *out++ = lowpass;

        output *= Q;  //scale the feedback
}
```

Time domain convolution with O(n^log2(3)) (click this to go back to the index)

References : Wilfried Welti

Notes :
[Quoted from Wilfrieds mail...]
I found last weekend that it is possible to do convolution in time domain (no complex numbers, 100% exact result with int) with O(n^log2(3)) (about O(n^1.58)).

Due to smaller overhead compared to FFT-based convolution, it should be the fastest algorithm for medium sized FIR's.
Though, it's slower as FFT-based convolution for large n.

It's pretty easy:

Let's say we have two finite signals of length 2n, which we want convolve : A and B. Now we split both signals into parts of size n, so we get A = A1 + A2, and B = B1 +B2.

Now we can write:

(1) A*B = (A1+A2)*(B1+B2) = A1*B1 + A2*B1 + A1*B2 + A2*B2

where * means convolution.

This we knew already: We can split a convolution into four convolutions of halved size.

Things become interesting when we start shifting blocks in time:

Be z a signal which has the value 1 at x=1 and zero elsewhere. Convoluting a signal X with z is equivalent to shifting X by one rightwards. When I define z^n as n-fold convolution of z with itself, like: z^1 = z, z^2 = z*z, z^0 = z shifted leftwards by 1 = impulse at x=0, and so on, I can use it to shift signals:

X * z^n means shifting the signal X by the value n rightwards.
X * z^-n means shifting the signal X by the value n leftwards.

Now we look at the following term:

(2) (A1 + A2 * z^-n) * (B1 + B2 * z^-n)

This is a convolution of two blocks of size n: We shift A2 by n leftwards so it completely overlaps A1, then we add them.
We do the same thing with B1 and B2. Then we convolute the two resulting blocks.

now let's transform this term:

(3) (A1 + A2 * z^-n) * (B1 + B2 * z^-n)

= A1*B1 + A1*B2*z^-n + A2*z^-n*B1 + A2*z^ n*B2*z^-n
= A1*B1 + (A1*B2 + A2*B1)*z^-n + A2*B2*z^-2n

(4) (A1 + A2 * z^-n) * (B1 + B2 * z^-n) - A1*B1 - A2*B2*z^-2n

= (A1*B2 + A2*B1)*z^-n

Now we convolute both sides of the equation (4) by z^n:

(5) (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n - A1*B1*z^n - A2*B2*z^-n

= (A1*B2 + A2*B1)

Now we see that the right part of equation (5) appears within equation (1), so we can replace this appearance by the left part of eq (5).

(6) A*B = (A1+A2)*(B1+B2) = A1*B1 + A2*B1 + A1*B2 + A2*B2

= A1*B1
+ (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n - A1*B1*z^n - A2*B2*z^-n
+ A2*B2

Voila!

We have constructed the convolution of A*B with only three convolutions of halved size. (Since the convolutions with z^n and z^-n are only shifts of blocks with size n, they of course need only n operations for processing :)

This can be used to construct an easy recursive algorithm of Order O(n^log2(3))

Code :
```
void convolution(value* in1, value* in2, value* out, value* buffer, int size)
{
  value* temp1 = buffer;
  value* temp2 = buffer + size/2;
  int i;

  // clear output.
  for (i=0; i<size*2; i++) out[i] = 0;

  // Break condition for recursion: 1x1 convolution is multiplication.
```

```
  if (size == 1)
  {
    out[0] = in1[0] * in2[0];
    return;
  }

  // first calculate (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n

  signal_add(in1, in1+size/2, temp1, size/2);
  signal_add(in2, in2+size/2, temp2, size/2);
  convolution(temp1, temp2, out+size/2, buffer+size, size/2);

  // then add A1*B1 and substract A1*B1*z^n

  convolution(in1, in2, temp1, buffer+size, size/2);
  signal_add_to(out, temp1, size);
  signal_sub_from(out+size/2, temp1, size);

  // then add A2*B2 and substract A2*B2*z^-n

  convolution(in1+size/2, in2+size/2, temp1, buffer+size, size/2);
  signal_add_to(out+size, temp1, size);
  signal_sub_from(out+size/2, temp1, size);
}
```

"value" may be a suitable type like int or float.
Parameter "size" is the size of the input signals and must be a power of 2. out and buffer must point to arrays of size 2*n.

Just to be complete, the helper functions:

```
void signal_add(value* in1, value* in2, value* out, int size)
{
  int i;
  for (i=0; i<size; i++) out[i] = in1[i] + in2[i];
}

void signal_sub_from(value* out, value* in, int size)
{
  int i;
  for (i=0; i<size; i++) out[i] -= in[i];
}

void signal_add_to(value* out, value* in, int size)
{
  int i;
  for (i=0; i<size; i++) out[i] += in[i];
}
```

Comments
  from : Christian@savioursofsoul.de
  comment : Here is a delphi translation of the code:

```
// "value" may be a suitable type like int or float.
// Parameter "size" is the size of the input signals and must be a power of 2.
// out and buffer must point to arrays of size 2*n.

procedure signal_add(in1, in2, ou1 :PValue; Size:Integer);
var i           : Integer;
begin
 for i:=0 to Size-1 do
  begin
   ou1^[i] := in1^[i] + in2^[i];
  end;
end;

procedure signal_sub_from(in1, ou1 :PValue; Size:Integer);
var i       : Integer;
begin
 for i:=0 to Size-1 do
  begin
   ou1^[i] := ou1^[i] - in1^[i];
  end;
end;

procedure signal_add_to(in1, ou1: PValue; Size:Integer);
var i       : Integer;
    po, pi1 : PValue;
begin
 po:=ou1;
 pi1:=in1;
 for i:=0 to Size-1 do
  begin
   ou1^[i] := ou1^[i] + in1^[i];
   Inc(po);
   Inc(pi1);
  end;
end;
```

```
procedure convolution(in1, in2, ou1, buffer :PValue; Size:Integer);
var tmp1, tmp2 : PValue;
    i          : Integer;
begin
 tmp1:=Buffer;
 tmp2:=@(Buffer^[(Size div 2)]);

 // clear output.
 for i:=0 to size*2 do ou1^[i]:=0;

 // Break condition for recursion: 1x1 convolution is multiplication.
 if Size = 1 then
  begin
   ou1^[0] := in1^[0] * in2^[0];
   exit;
  end;

 // first calculate (A1 + A2 * z^-n)*(B1 + B2 * z^-n)*z^n
 signal_add(in1, @(in1^[(Size div 2)]), tmp1, Size div 2);
 signal_add(in2, @(in1^[(Size div 2)]), tmp2, Size div 2);
 convolution(tmp1, tmp2, @(ou1^[(Size div 2)]), @(Buffer^[Size]), Size div 2);

 // then add A1*B1 and substract A1*B1*z^n
 convolution(in1, in2, tmp1, @(Buffer^[Size]), Size div 2);
 signal_add_to(ou1, tmp1, size);
 signal_sub_from(@(ou1^[(Size div 2)]), tmp1, size);

 // then add A2*B2 and substract A2*B2*z^-n
 convolution(@(in1^[(Size div 2)]), @(in2^[(Size div 2)]), tmp1, @(Buffer^[Size]), Size div 2);
 signal_add_to(@(ou1^[Size]), tmp1, size);
 signal_sub_from(@(ou1^[Size]), tmp1, size);
end;
```

<u>from</u> : Christian@savioursofsoul.de
<u>comment</u> : Sorry, i forgot the definitions:

```
type
  Values = Array[0..0] of Single;
  PValue = ^Values;
```

Time domain convolution with O(n^log2(3)) (click this to go back to the index)

References : Posted by Magnus Jonsson

Notes :
[see other code by Wilfried Welti too!]

Code :
```
void mul_brute(float *r, float *a, float *b, int w)
{
    for (int i = 0; i < w+w; i++)
        r[i] = 0;
    for (int i = 0; i < w; i++)
    {
        float *rr = r+i;
        float ai = a[i];
        for (int j = 0; j < w; j++)
            rr[j] += ai*b[j];
    }
}

// tmp must be of length 2*w
void mul_knuth(float *r, float *a, float *b, int w, float *tmp)
{
    if (w < 30)
    {
        mul_brute(r, a, b, w);
    }
    else
    {
        int m = w>>1;

        for (int i = 0; i < m; i++)
        {
            r[i  ] = a[m+i]-a[i  ];
            r[i+m] = b[i  ]-b[m+i];
        }

        mul_knuth(tmp, r  , r+m, m, tmp+w);
        mul_knuth(r  , a  , b  , m, tmp+w);
        mul_knuth(r+w, a+m, b+m, m, tmp+w);

        for (int i = 0; i < m; i++)
        {
            float bla = r[m+i]+r[w+i];
            r[m+i] = bla+r[i    ]+tmp[i  ];
            r[w+i] = bla+r[w+m+i]+tmp[i+m];
        }
    }
}
```

Various Biquad filters (click this to go back to the index)

References : JAES, Vol. 31, No. 11, 1983 November

Linked file : filters003.txt (this linked file is included below)

Notes :
(see linkfile)
Filters included are:
presence
shelvelowpass
2polebp
peaknotch
peaknotch2

Linked files
```
/*
 * Presence and Shelve filters as given in
 *   James A. Moorer
 *   The manifold joys of conformal mapping:
 *   applications to digital filtering in the studio
 *   JAES, Vol. 31, No. 11, 1983 November
 */

#define SPN MINDOUBLE

double bw2angle(a,bw)
double a,bw;
{
  double T,d,sn,cs,mag,delta,theta,tmp,a2,a4,asnd;

  T = tan(2.0*PI*bw);
  a2 = a*a;
  a4 = a2*a2;
  d = 2.0*a2*T;
  sn = (1.0 + a4)*T;
  cs = (1.0 - a4);
  mag = sqrt(sn*sn + cs*cs);
  d /= mag;
  delta = atan2(sn,cs);
  asnd = asin(d);
  theta = 0.5*(PI - asnd - delta);
  tmp = 0.5*(asnd-delta);
  if ((tmp > 0.0) && (tmp < theta)) theta = tmp;
  return(theta/(2.0*PI));
}

void presence(cf,boost,bw,a0,a1,a2,b1,b2)
double cf,boost,bw,*a0,*a1,*a2,*b1,*b2;
{
  double a,A,F,xfmbw,C,tmp,alphan,alphad,b0,recipb0,asq,F2,a2plus1,ma2plus1;

  a = tan(PI*(cf-0.25));
  asq = a*a;
  A = pow(10.0,boost/20.0);
  if ((boost < 6.0) && (boost > -6.0)) F = sqrt(A);
  else if (A > 1.0) F = A/sqrt(2.0);
  else F = A*sqrt(2.0);
  xfmbw = bw2angle(a,bw);

  C = 1.0/tan(2.0*PI*xfmbw);
  F2 = F*F;
  tmp = A*A - F2;
  if (fabs(tmp) <= SPN) alphad = C;
  else alphad = sqrt(C*C*(F2-1.0)/tmp);
  alphan = A*alphad;

  a2plus1 = 1.0 + asq;
  ma2plus1 = 1.0 - asq;
  *a0 = a2plus1 + alphan*ma2plus1;
  *a1 = 4.0*a;
  *a2 = a2plus1 - alphan*ma2plus1;

  b0 = a2plus1 + alphad*ma2plus1;
  *b2 = a2plus1 - alphad*ma2plus1;

  recipb0 = 1.0/b0;
  *a0 *= recipb0;
  *a1 *= recipb0;
  *a2 *= recipb0;
  *b1 = *a1;
  *b2 *= recipb0;
```

```
}

void shelve(cf,boost,a0,a1,a2,b1,b2)
double cf,boost,*a0,*a1,*a2,*b1,*b2;
{
  double a,A,F,tmp,b0,recipb0,asq,F2,gamma2,siggam2,gam2p1;
  double gamman,gammad,ta0,ta1,ta2,tb0,tb1,tb2,aa1,ab1;

  a = tan(PI*(cf-0.25));
  asq = a*a;
  A = pow(10.0,boost/20.0);
  if ((boost < 6.0) && (boost > -6.0)) F = sqrt(A);
  else if (A > 1.0) F = A/sqrt(2.0);
  else F = A*sqrt(2.0);

  F2 = F*F;
  tmp = A*A - F2;
  if (fabs(tmp) <= SPN) gammad = 1.0;
  else gammad = pow((F2-1.0)/tmp,0.25);
  gamman = sqrt(A)*gammad;

  gamma2 = gamman*gamman;
  gam2p1 = 1.0 + gamma2;
  siggam2 = 2.0*sqrt(2.0)/2.0*gamman;
  ta0 = gam2p1 + siggam2;
  ta1 = -2.0*(1.0 - gamma2);
  ta2 = gam2p1 - siggam2;

  gamma2 = gammad*gammad;
  gam2p1 = 1.0 + gamma2;
  siggam2 = 2.0*sqrt(2.0)/2.0*gammad;
  tb0 = gam2p1 + siggam2;
  tb1 = -2.0*(1.0 - gamma2);
  tb2 = gam2p1 - siggam2;

  aa1 = a*ta1;
  *a0 = ta0 + aa1 + asq*ta2;
  *a1 = 2.0*a*(ta0+ta2)+(1.0+asq)*ta1;
  *a2 = asq*ta0 + aa1 + ta2;

  ab1 = a*tb1;
  b0 = tb0 + ab1 + asq*tb2;
  *b1 = 2.0*a*(tb0+tb2)+(1.0+asq)*tb1;
  *b2 = asq*tb0 + ab1 + tb2;

  recipb0 = 1.0/b0;
  *a0 *= recipb0;
  *a1 *= recipb0;
  *a2 *= recipb0;
  *b1 *= recipb0;
  *b2 *= recipb0;
}

void initfilter(f)
filter *f;
{
  f->x1 = 0.0;
  f->x2 = 0.0;
  f->y1 = 0.0;
  f->y2 = 0.0;
  f->y = 0.0;
}

void setfilter_presence(f,freq,boost,bw)
filter *f;
double freq,boost,bw;
{
  presence(freq/(double)SR,boost,bw/(double)SR,
           &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
  f->cy1 = -f->cy1;
  f->cy2 = -f->cy2;
}

void setfilter_shelve(f,freq,boost)
filter *f;
double freq,boost;
{
  shelve(freq/(double)SR,boost,
   &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
  f->cy1 = -f->cy1;
  f->cy2 = -f->cy2;
}
```

```
void setfilter_shelvelowpass(f,freq,boost)
filter *f;
double freq,boost;
{
  double gain;

  gain = pow(10.0,boost/20.0);
  shelve(freq/(double)SR,boost,
   &f->cx,&f->cx1,&f->cx2,&f->cy1,&f->cy2);
  f->cx /= gain;
  f->cx1 /= gain;
  f->cx2 /= gain;
  f->cy1 = -f->cy1;
  f->cy2 = -f->cy2;
}

/*
 * As in ''An introduction to digital filter theory'' by Julius O. Smith
 * and in Moore's book; I use the normalized version in Moore's book.
 */
void setfilter_2polebp(f,freq,R)
filter *f;
double freq,R;
{
  double theta;

  theta = 2.0*PI*freq/(double)SR;
  f->cx = 1.0-R;
  f->cx1 = 0.0;
  f->cx2 = -(1.0-R)*R;
  f->cy1 = 2.0*R*cos(theta);
  f->cy2 = -R*R;
}

/*
 * As in
 *   Stanley A. White
 *   Design of a digital biquadratic peaking or notch filter
 *   for digital audio equalization
 *   JAES, Vol. 34, No. 6, 1986 June
 */
void setfilter_peaknotch(f,freq,M,bw)
filter *f;
double freq,M,bw;
{
  double w0,p,om,ta,d;

  w0 = 2.0*PI*freq;
  if ((1.0/sqrt(2.0) < M) && (M < sqrt(2.0))) {
    fprintf(stderr,"peaknotch filter: 1/sqrt(2) < M < sqrt(2)\n");
    exit(-1);
  }
  if (M <= 1.0/sqrt(2.0)) p = sqrt(1.0-2.0*M*M);
  if (sqrt(2.0) <= M) p = sqrt(M*M-2.0);
  om = 2.0*PI*bw;
  ta = tan(om/((double)SR*2.0));
  d = p+ta;
  f->cx = (p+M*ta)/d;
  f->cx1 = -2.0*p*cos(w0/(double)SR)/d;
  f->cx2 = (p-M*ta)/d;
  f->cy1 = 2.0*p*cos(w0/(double)SR)/d;
  f->cy2 = -(p-ta)/d;
}

/*
 * Some JAES's article on ladder filter.
 * freq (Hz), gdb (dB), bw (Hz)
 */
void setfilter_peaknotch2(f,freq,gdb,bw)
filter *f;
double freq,gdb,bw;
{
  double k,w,bwr,abw,gain;

  k = pow(10.0,gdb/20.0);
  w = 2.0*PI*freq/(double)SR;
  bwr = 2.0*PI*bw/(double)SR;
  abw = (1.0-tan(bwr/2.0))/(1.0+tan(bwr/2.0));
  gain = 0.5*(1.0+k+abw-k*abw);
  f->cx = 1.0*gain;
  f->cx1 = gain*(-2.0*cos(w)*(1.0+abw))/(1.0+k+abw-k*abw);
  f->cx2 = gain*(abw+k*abw+1.0-k)/(abw-k*abw+1.0+k);
  f->cy1 = 2.0*cos(w)/(1.0+tan(bwr/2.0));
```

```
  f->cy2 = -abw;
}

double applyfilter(f,x)
filter *f;
double x;
{
  f->x = x;
  f->y = f->cx * f->x + f->cx1 * f->x1 + f->cx2 * f->x2
    + f->cy1 * f->y1 + f->cy2 * f->y2;
  f->x2 = f->x1;
  f->x1 = f->x;
  f->y2 = f->y1;
  f->y1 = f->y;
  return(f->y);
}
```

Zoelzer biquad filters (click this to go back to the index)

Type : biquad IIR

References : Udo Zoelzer: Digital Audio Signal Processing (John Wiley & Sons, ISBN 0 471 97226 6), Chris Townsend

Notes :
Here's the formulas for the Low Pass, Peaking, and Low Shelf, which should
cover the basics. I tried to convert the formulas so they are little more consistent.
Also, the Zolzer low pass/shelf formulas didn't have adjustable Q, so I added that for
consistency with Roberts formulas as well. I think someone may want to check that I did
it right.
------------ Chris Townsend
I mistranscribed the low shelf cut formulas.
Hopefully this is correct. Thanks to James McCartney for noticing.
------------ Chris Townsend

Code :
```
omega = 2*PI*frequency/sample_rate

K=tan(omega/2)
Q=Quality Factor
V=gain

LPF:   b0 =  K^2
       b1 =  2*K^2
       b2 =  K^2
       a0 =  1 + K/Q + K^2
       a1 =  2*(K^2 - 1)
       a2 =  1 - K/Q + K^2

peakingEQ:
      boost:
      b0 =  1 + V*K/Q + K^2
      b1 =  2*(K^2 - 1)
      b2 =  1 - V*K/Q + K^2
      a0 =  1 + K/Q + K^2
      a1 =  2*(K^2 - 1)
      a2 =  1 - K/Q + K^2

      cut:
      b0 =  1 + K/Q + K^2
      b1 =  2*(K^2 - 1)
      b2 =  1 - K/Q + K^2
      a0 =  1 + V*K/Q + K^2
      a1 =  2*(K^2 - 1)
      a2 =  1 - V*K/Q + K^2

lowShelf:
     boost:
       b0 =  1 + sqrt(2*V)*K + V*K^2
       b1 =  2*(V*K^2 - 1)
       b2 =  1 - sqrt(2*V)*K + V*K^2
       a0 =  1 + K/Q + K^2
       a1 =  2*(K^2 - 1)
       a2 =  1 - K/Q + K^2

     cut:
       b0 =  1 + K/Q + K^2
       b1 =  2*(K^2 - 1)
       b2 =  1 - K/Q + K^2
       a0 =  1 + sqrt(2*V)*K + V*K^2
       a1 =  2*(v*K^2 - 1)
       a2 =  1 - sqrt(2*V)*K + V*K^2
```

Comments
from : signalzerodb@yahoo.com
comment : I get a different result for the low-shelf boost with parametric control.
Zolzer builds his lp shelf from a pair of poles and a pair of zeros at:
poles = Q(-1 +- j)
zeros = sqrt(V)Q(-1 +- j)
Where (in the book) Q=1/sqrt(2)
So,

```
      s^2 + 2sqrt(V)Qs + 2VQ^2
H(s) = ------------------------
         s^2 + 2Qs +2Q^2
```

If you analyse this in terms of:
H(s) = LPF(s) + 1, it sort of falls apart, as we've gained a zero in the LPF. (as does zolzers)

Then, if we bilinear transform that, we get:

a0= 1 + 2*sqrt(V)*Q*K + 2*V*Q^2*K^2
a1= 2 (2*V*Q^2*K^2 - 1)
a2= 1 - 2*sqrt(V)*Q*K + 2*V*Q^2*K^2
b0= 1 + 2*Q*K + 2*Q^2*K^2
b1= 2 (2*Q^2*K^2 - 1)
b2= 1 - 2*Q*K + 2*Q^2*K^2

For:
H(z) = a0z^2 + a1z +a2 / b0z^2 + b1z + b2

Which, i /think/ is right...

Dave.

from : signalzerodb@yahoo.com
comment : Very sorry, I interpreted Zolzer's s-plane poles as z-plane poles. Too much digital stuff.

After getting back to grips with s-plane maths :) and much graphing to test that it's right, I still get slightly different results.

b0 = 1 + sqrt(V)*K/Q + V*K^2
b1 = 2*(V*K^2 - 1)
b2 = 1 - sqrt(V)*K/Q + V*K^2
a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2
The way the filter works is to have two poles on a unit circle around the origin in the s-plane, and two zeros that start at the poles at V0=1, and move outwards. The above co-efficients represent that. Chris's original results put the poles in the right place, but put the zeros at the location where the poles would be if they were butterworth, and move out from there - yielding some rather strange results...

But I've graphed that extensively, and it works fine now :)

Dave.

2 Wave shaping things (click this to go back to the index)

References : Posted by Frederic Petrot

Notes :
Makes nice saturations effects that can be easilly computed using cordic
First using a atan function:
y1 using k=16
max is the max value you can reach (32767 would be a good guess)
Harmonics scale down linealy and not that fast

Second using the hyperbolic tangent function:
y2 using k=2
Harmonics scale down linealy very fast

Code :
```
y1 = (max>>1) * atan(k * x/max)

y2 = max * th(x/max)
```

Alien Wah (click this to go back to the index)

References : Nasca Octavian Paul (paulnasca[AT]email.ro)

Linked file : alienwah.c (this linked file is included below)

Notes :
"I found this algoritm by "playing around" with complex numbers. Please email me your opinions about it.

Paul."

Comments
from : ignatz@webmail.co.za
comment : need help porting this alienwah to C, i'm running linux d;>

from : antiprosynthesis@hotmail.com
comment : Where to download the complex.h you included?

Linked files
```
/*
 Alien-Wah by Nasca Octavian Paul from Tg. Mures, Romania
 e-mail:  <paulnasca@email.ro> or <paulnasca@yahoo.com>.
*/

/*
 The algorithm was found by me by mistake(I was looking for something else);
 I called this effect "Alien Wah" because sounds a bit like wahwah, but more strange.
 The ideea of this effect is very simple: It is a feedback delay who uses complex numbers.
 If x[] represents the input and y[] is the output, so a simple feedback delay looks like this:
 y[n]=y[n-delay]*fb+x[n]*(1-fb)

 'fb' is a real number between 0 and 1.
 If you change the fb with a complex number who has the MODULUS smaller than 1, it will look like
this.

 fb=R*(cos(alpha)+i*sin(alpha));  i^2=-1; R<1;
 y[n]=y[n-delay]*R*(cos(alpha)+i*sin(alpha))+x[n]*(1-R);

 alpha is the phase of the number and is controlled by the LFO(Low Frequency Oscillator).
 If the 'delay' parameter is low, the effect sounds more like wah-wah,
 but if it is big, the effect will sound very interesting.
 The input x[n] has the real part of the samples from the wavefile and the imaginary part is zero.
 The output of this effect is the real part of y[n].

 Here it is a simple and unoptimised implementation of the effect. All parameters should be
changed at compile time.
 It was tested only with Borland C++ 3.1.

 Please send me your opinions about this effect.
 Hope you like it (especially if you are play to guitar).
 Paul.
*/

/*
Alien Wah Parameters

 freq       - "Alien Wah" LFO frequency
 startphase - "Alien Wah" LFO startphase (radians), needed for stereo
 fb         - "Alien Wah" FeedBack (0.0 - low feedback, 1.0 = 100% high feedback)
 delay      -  delay in samples at 44100 KHz (recomanded from 5 to 50...)
*/

#include <complex.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <math.h>

/*
 .raw files are raw files (without header), signed 16 bit,mono
*/
#define infile "a.raw" //input file
#define outfile "b.raw" //input file
#define samplerate 44100

#define bufsize 1024
int buf1[bufsize];//input buffer
int buf2[bufsize];//output buffer
```

```
#define lfoskipsamples 25 // How many samples are processed before compute the lfo value again

struct params
{
   float freq,startphase,fb;
   int delay;
} awparams;
//alien wah internal parameters

struct alienwahinternals
{
 complex *delaybuf;
 float lfoskip;
 long int t;
 complex c;
 int k;
} awint;

//effect initialisation
void init(float freq,float startphase,float fb,int delay){
  awparams.freq=freq;
  awparams.startphase=startphase;
  awparams.fb=fb/4+0.74;
  awparams.delay=(int)(delay/44100.0*samplerate);
  if (delay<1) delay=1;
  awint.delaybuf=new complex[awparams.delay];
  int i;
  for (i=0;i<delay;i++) awint.delaybuf[i]=complex(0,0);
  awint.lfoskip=freq*2*3.141592653589/samplerate;
  awint.t=0;
}

//process buffer
void process()
{
 int i;
 float lfo,out;
 complex outc;
 for(i=0;i<bufsize;i++)
 {
   if (awint.t++%lfoskipsamples==0)
   {
      lfo=(1+cos(awint.t*awint.lfoskip+awparams.startphase));
      awint.c=complex(cos(lfo)*awparams.fb,sin(lfo)*awparams.fb);
   };
   outc=awint.c*awint.delaybuf[awint.k]+(1-awparams.fb)*buf1[i];
   awint.delaybuf[awint.k]=outc;
   if ((++awint.k)>=awparams.delay)
      awint.k=0;
   out=real(outc)*3;  //take real part of outc
   if (out<-32768) out=-32768;
   else if (out>32767) out=32767; //Prevents clipping
   buf2[i]=out;
 };
}

int main()
{
  char f1,f2;
  int readed;
  long int filereaded=0;
  printf("\n");
  f1=open(infile,O_RDONLY|O_BINARY);
  remove(outfile);
  f2=open(outfile,O_BINARY|O_CREAT,S_IWRITE);
  long int i;

  init(0.6,0,0.5,20);  //effects parameters

  do
  {
    readed=read(f1,buf1,bufsize*2);

    process();

    write(f2,buf2,readed);
    printf("%ld  bytes \r",filereaded);
    filereaded+=readed;
  }while (readed==bufsize*2);

  delete(awint.delaybuf);
```

```
  close(f1);
  close(f2);
  printf("\n\n");

  return(0);
}
```

Bit quantization/reduction effect (click this to go back to the index)

Type : Bit-level noise-generating effect

References : Posted by Jon Watte

Notes :
This function, run on each sample, will emulate half the effect of running your signal through a Speak-N-Spell or similar low-bit-depth circuitry.

The other half would come from downsampling with no aliasing control, i e replicating every N-th sample N times in the output signal.

Code :
```
short keep_bits_from_16( short input, int keepBits ) {
  return (input & (-1 << (16-keepBits)));
}
```

Class for waveguide/delay effects (click this to go back to the index)

Type : IIR filter

References : Posted by arguru[AT]smartelectronix.com

Notes :
Flexible-time, non-sample quantized delay , can be used for stuff like waveguide synthesis or time-based (chorus/flanger) fx.

MAX_WG_DELAY is a constant determining MAX buffer size (in samples)

Code :
```
class cwaveguide
{
public:
 cwaveguide(){clear();}
 virtual ~cwaveguide(){};

 void clear()
 {
  counter=0;
  for(int s=0;s<MAX_WG_DELAY;s++)
   buffer[s]=0;
 }

 inline float feed(float const in,float const feedback,double const delay)
 {
  // calculate delay offset
  double back=(double)counter-delay;

  // clip lookback buffer-bound
  if(back<0.0)
   back=MAX_WG_DELAY+back;

  // compute interpolation left-floor
  int const index0=floor_int(back);

  // compute interpolation right-floor
  int index_1=index0-1;
  int index1=index0+1;
  int index2=index0+2;

  // clip interp. buffer-bound
  if(index_1<0)index_1=MAX_WG_DELAY-1;
  if(index1>=MAX_WG_DELAY)index1=0;
  if(index2>=MAX_WG_DELAY)index2=0;

  // get neighbourgh samples
  float const y_1= buffer [index_1];
  float const y0 = buffer [index0];
  float const y1 = buffer [index1];
  float const y2 = buffer [index2];

  // compute interpolation x
  float const x=(float)back-(float)index0;

  // calculate
  float const c0 = y0;
  float const c1 = 0.5f*(y1-y_1);
  float const c2 = y_1 - 2.5f*y0 + 2.0f*y1 - 0.5f*y2;
  float const c3 = 0.5f*(y2-y_1) + 1.5f*(y0-y1);

  float const output=((c3*x+c2)*x+c1)*x+c0;

  // add to delay buffer
  buffer[counter]=in+output*feedback;

  // increment delay counter
  counter++;

  // clip delay counter
  if(counter>=MAX_WG_DELAY)
   counter=0;

  // return output
  return output;
 }

 float buffer[MAX_WG_DELAY];
 int  counter;
};
```

Decimator (click this to go back to the index)

Type : Bit-reducer and sample&hold unit

References : Posted by tobyear[AT]web[DOT]de

Notes :
This is a simple bit and sample rate reduction code, maybe some of you can use it. The parameters are bits (1..32) and rate (0..1, 1 is the original samplerate).
Call the function like this:
y=decimate(x);

A VST plugin implementing this algorithm (with full Delphi source code included) can be downloaded from here:
http://tobybear.phreque.com/decimator.zip

Comments/suggestions/improvements are welcome, send them to: tobybear@web.de

Code :
```
// bits: 1..32
// rate: 0..1 (1 is original samplerate)

********** Pascal source **********
var m:longint;
    y,cnt,rate:single;

// call this at least once before calling
// decimate() the first time
procedure setparams(bits:integer;shrate:single);
begin
 m:=1 shl (bits-1);
 cnt:=1;
 rate:=shrate;
end;

function decimate(i:single):single;
begin
 cnt:=cnt+rate;
 if (cnt>1) then
 begin
  cnt:=cnt-1;
  y:=round(i*m)/m;
 end;
 result:=y;
end;

********** C source **********
int bits=16;
float rate=0.5;

long int m=1<<(bits-1);
float y=0,cnt=0;

float decimate(float i)
{
 cnt+=rate;
 if (cnt>=1)
 {
  cnt-=1;
  y=(long int)(i*m)/(float)m;
 }
 return y;
}
```

Comments
from : kaleja@estarcion.com
comment : Nothing wrong with that, but you can also do fractional-bit-depth decimations, allowing smooth degradation from high bit depth to
low and back:

```
// something like this -- this is
// completely off the top of my head
// precalculate the quantization level
float bits; // effective bit depth
float quantum = powf( 2.0f, bits );

// per sample
y = floorf( x * quantum ) / quantum;
```

----

from : dr.kef@spray.se
comment : it looks to me like the c-line

long int m=1<<(bits-1);

doesnt give the correct number of quantisation levels if the number of levels is defined as 2^bits. if bits=2 for instance, the above code line returns a bit pattern of 10 (3) and not 11 (2^2) like one would expect.

please, do correct me if im wrong.

/heatrof

from : resofactor@hotmail.com
comment : just getting into coding, i've mainly been working with synthedit...but would really like to move on into the bigger arena? any pointers for a DSP newbie-totally not hip on structured programming...yet! :O)

Delay time calculation for reverberation (click this to go back to the index)

References : Posted by Andy Mucho

Notes :
This is from some notes I had scribbled down from a while back on automatically calculating diffuse delays. Given an intial delay line gain and time, calculate the times and feedback gain for numlines delay lines..

Code :
```
int   numlines = 8;
float t1 = 50.0;        // d0 time
float g1 = 0.75;        // d0 gain
float rev = -3*t1 / log10 (g1);

for (int n = 0; n < numlines; ++n)
{
  float dt = t1 / pow (2, (float (n) / numlines));
  float g = pow (10, -((3*dt) / rev));
  printf ("d%d t=%.3f g=%.3f\n", n, dt, g);
}

The above with t1=50.0 and g1=0.75 yields:

 d0 t=50.000 g=0.750
 d1 t=45.850 g=0.768
 d2 t=42.045 g=0.785
 d3 t=38.555 g=0.801
 d4 t=35.355 g=0.816
 d5 t=32.421 g=0.830
 d6 t=29.730 g=0.843
 d7 t=27.263 g=0.855

To go more diffuse, chuck in dual feedback paths with a one cycle delay
effectively creating a phase-shifter in the feedback path, then things get
more exciting.. Though what the optimum phase shifts would be I couldn't
tell you right now..
```

Early echo's with image-mirror technique (click this to go back to the index)

References : Donald Schulz

Linked file : early_echo.c (this linked file is included below)

Linked file : early_echo_eng.c (this linked file is included below)

Notes :
(see linked files)
Donald Schulz's code for computing early echoes using the image-mirror method. There's an english and a german version.

Linked files

```
/*
From: "Donald Schulz" <d.schulz@gmx.de>
To: <music-dsp@shoko.calarts.edu>
Subject: [music-dsp] Image-mirror method source code
Date: Sun, 11 Jun 2000 15:01:51 +0200

A while ago I wrote a program to calculate early echo responses.
As there seems to be some interest in this, I now post it into the
public domain.

Have phun,

Donald.
*/

/****************************************************************************
 *
 * Early Echo Computation using image-mirror method
 *
 * Position of listener, 2 sound-sources, room-size may be set.
 * Four early echo responses are calculated (from left sound source and
 * right sound source to left and right ear). Angle with which the sound
 * meets the ears is taken into account.
 * The early echo response from left sound source to left ear is printed
 * to screen for demonstration, the first table contains the delay times
 * and the second one the weights.
 *
 * Program is released into the public domain.
 *
 * Sorry for german comments :-(
 * Some frequently used german words:
 * hoerpos : listening position
 * breite : width
 * laenge : length
 * hoehe : height
 * daempfung : damping
 * links : left
 * rechts : right
 * Ohr    : ear
 * Winkel : angle
 * Wichtung : weight
 * Normierung : normalization
 *
 *
 * If someone does some improvements on this, I (Donald, d.schulz@gmx.de)
 * would be happy to get the improved code.
 *
 ***************************************************************************/



#include <math.h>
#include <stdio.h>

#define early_length 0x4000     /* Laenge der Puffer fuer early-echo */
#define early_length_1 0x3fff

#define early_tap_num 20        /* Anzahl an early-echo taps */

#define breite 15.0              /* 15 m breiter Raum (x)*/
#define laenge 20.0             /* 20 m lang (y) */
#define hoehe 10.0               /* 10 m hoch (z)*/
#define daempfung 0.999         /* Daempfungsfaktor bei Reflexion */

#define hoerposx 7.91            /* hier sitzt der Hoerer (linkes Ohr) */
#define hoerposy 5.0
#define hoerposz 2.0

#define leftposx 5.1            /* hier steht die linke Schallquelle */
```

```
#define leftposy 16.3
#define leftposz 2.5

#define rightposx 5.9            /* hier steht die rechte Schallquelle */
#define rightposy 6.3
#define rightposz 1.5

#define i_length 32             /* Laenge des Eingangs-Zwischenpuffers */
#define i_length_1 31
#define o_length 32             /* Laenge des Ausgangs-Zwischenpuffers */
#define o_length_1 31


float *early_l2l;  /* linker Kanal nach linkem Ohr */
float *early_l2r;  /* linker Kanal nach rechtem Ohr */
float *early_r2l;  /* rechter Kanal nach linkem Ohr */
float *early_r2r;  /* rechter Kanal nach rechtem Ohr */

int early_pos=0;

int e_delays_l2l[early_tap_num];       /* Delays der early-echos */
float e_values_l2l[early_tap_num];     /* Gewichtungen der delays */
int e_delays_l2r[early_tap_num];
float e_values_l2r[early_tap_num];
int e_delays_r2l[early_tap_num];
float e_values_r2l[early_tap_num];
int e_delays_r2r[early_tap_num];
float e_values_r2r[early_tap_num];


/* Early-echo Berechnung mittels Spiegelquellenverfahren

Raummodell:
H - Hoererposition
L - Spiegelschallquellen des linken Kanales
U - Koordinatenursprung
Raum sei 11 meter breit und 5 meter lang (1 Zeichen = 1 meter)
Linker Kanal stehe bei x=2  y=4  z=?
Hoerer stehe bei x=5 y=1 z=?
___________________________________________________________________
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
__L___L________________L___L________________L___L________________L_
L | L        |       L | L       |        L | L       |        L|
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |    H    |          |         |         |
_________________________X_________________________________________
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
__L___L________________L___L_________________L__L________________L_
*/

main()
{
  int i,j,select;
  float dist_max;
  float x,y,z,xref,yref,zref;
  float x_pos,y_pos,z_pos;
  float distance,winkel;
  float wichtung;
  float normierungr,normierungl;


  early_l2l=(float *)malloc(early_length*sizeof(float));
  early_l2r=(float *)malloc(early_length*sizeof(float));;
  early_r2l=(float *)malloc(early_length*sizeof(float));;
  early_r2r=(float *)malloc(early_length*sizeof(float));;


  /* Erst mal Echos loeschen: */
  for (i=0;i<early_length;i++)
  early_l2l[i]=early_l2r[i]=early_r2l[i]=early_r2r[i]=0.0;

  dist_max=300.0*early_length/44100.0;  /* 300 m/s Schallgeschwindigkeit */

  /* Echo vom LINKEN Kanal auf linkes/rechtes Ohr berechnen */

  for (x=-ceil(dist_max/(2*laenge));x<=ceil(dist_max/(2*laenge));x++)
```

```
for (y=-ceil(dist_max/(2*breite));y<=ceil(dist_max/(2*breite));y++)
for (z=-ceil(dist_max/(2*hoehe));z<=ceil(dist_max/(2*hoehe));z++)
{
  xref=2*x*breite;
  yref=2*y*laenge;
  zref=2*z*hoehe;
  for (select=0;select<8;select++)   /* vollstaendige Permutation */
  {
    if (select&1) x_pos=xref+leftposx;
    else x_pos=xref-leftposx;
    if (select&2) y_pos=yref+leftposy;
    else y_pos=yref-leftposy;
    if (select&4) z_pos=zref+leftposz;
    else z_pos=zref-leftposz;
    /* Jetzt steht die absolute Position der Quelle in ?_pos */

    /* Relative Position zum linken Ohr des Hoerers bestimmen: */
    x_pos-=hoerposx;
    y_pos-=hoerposy;
    z_pos-=hoerposz;


    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
        winkel+=3.1415926/2;     wichtung = 1 - winkel/4;
      }
      else            /* Klang kommt von hinten: */
      {
        winkel-=3.1415926/2;  wichtung= 1 + winkel/4;
      }
      /* Early-echo gemaess Winkel und Entfernung gewichten: */
      early_l2l[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
    }

    /* Relative Position zum rechten Ohr des Hoerers bestimmen: */
    x_pos-=0.18;  /* Kopf ist 18 cm breit */

    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
        winkel-=3.1415926/2;     wichtung = 1 + winkel/4;
      }
      else            /* Klang kommt von hinten: */
      {
        winkel+=3.1415926/2;  wichtung= 1 - winkel/4;
      }
      /* Early-echo gemaess Winkel und Entfernung gewichten: */
      early_l2r[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
    }
  }
}

/* Echo vom RECHTEN Kanal auf linkes/rechtes Ohr berechnen */

for (x=-ceil(dist_max/(2*laenge));x<=ceil(dist_max/(2*laenge));x++)
for (y=-ceil(dist_max/(2*breite));y<=ceil(dist_max/(2*breite));y++)
for (z=-ceil(dist_max/(2*hoehe));z<=ceil(dist_max/(2*hoehe));z++)
{
  xref=2*x*breite;
  yref=2*y*laenge;
  zref=2*z*hoehe;
  for (select=0;select<8;select++)   /* vollstaendige Permutation */
  {
    if (select&1) x_pos=xref+rightposx;
    else x_pos=xref-rightposx;
    if (select&2) y_pos=yref+rightposy;
    else y_pos=yref-rightposy;
    if (select&4) z_pos=zref+rightposz;
    else z_pos=zref-rightposz;
    /* Jetzt steht die absolute Position der Quelle in ?_pos */

    /* Relative Position zum linken Ohr des Hoerers bestimmen: */
```

```
    x_pos-=hoerposx;
    y_pos-=hoerposy;
    z_pos-=hoerposz;


    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147.)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
        winkel+=3.1415926/2;     wichtung = 1 - winkel/4;
      }
      else            /* Klang kommt von hinten: */
      {
        winkel-=3.1415926/2;  wichtung= 1 + winkel/4;
      }
      /* Early-echo gemaess Winkel und Entfernung gewichten: */
      early_r2l[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
    }


    /* Und jetzt Early-Echo zweiter Kanal auf LINKES Ohr berechnen */
    x_pos-=0.18; /* Kopfbreite addieren */

    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallswinkel im Bereich -Pi/2 bis Pi/2 ermitteln: */
      winkel=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=links=>Verstaerkung=1 Pi=rechts=>Verstaerkung=0.22 (?) */
        winkel-=3.1415926/2;     wichtung = 1 + winkel/4;
      }
      else            /* Klang kommt von hinten: */
      {
        winkel+=3.1415926/2;  wichtung= 1 - winkel/4;
      }
      /* Early-echo gemaess Winkel und Entfernung gewichten: */
      early_r2r[(int) (distance*147.)]+=wichtung/(pow(distance,3.1));
    }
  }
}


/* Und jetzt aus berechnetem Echo die ersten early_tap_num Werte holen */
/* Erst mal e's zuruecksetzen: */
for (i=0;i<early_tap_num;i++)
{
  e_values_l2l[i]=e_values_l2r[i]=0.0;
  e_delays_l2l[i]=e_delays_l2r[i]=0;  /* Unangenehme Speicherzugriffe vermeiden */
  e_values_r2l[i]=e_values_r2r[i]=0.0;
  e_delays_r2l[i]=e_delays_r2r[i]=0;
}

/* und jetzt e_delays und e_values extrahieren: */
j=0;
normierungl=normierungr=0.0;
for(i=0;i<early_length;i++)
{
  if (early_l2l[i]!=0)
  {
    e_delays_l2l[j]=i;
    e_values_l2l[j]=early_l2l[i];
    normierungl+=early_l2l[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_l2r[i]!=0)
  {
    e_delays_l2r[j]=i;
    e_values_l2r[j]=early_l2r[i];
    normierungr+=early_l2r[i];
    j++;
  }
  if (j==early_tap_num) break;
```

```
  }
  j=0;
  for(i=0;i<early_length;i++)
  {
    if (early_r2l[i]!=0)
    {
      e_delays_r2l[j]=i;
      e_values_r2l[j]=early_r2l[i];
      normierungl+=early_r2l[i];
      j++;
    }
    if (j==early_tap_num) break;
  }
  j=0;
  for(i=0;i<early_length;i++)
  {
    if (early_r2r[i]!=0)
    {
      e_delays_r2r[j]=i;
      e_values_r2r[j]=early_r2r[i];
      normierungr+=early_r2r[i];
      j++;
    }
    if (j==early_tap_num) break;
  }

  /* groessere von beiden Normierungen verwenden: */
  if (normierungr>normierungl) normierungr=normierungl;

  for (j=0;j<early_tap_num;j++)
  {
    e_values_l2l[j]/=normierungr;
    e_values_l2r[j]/=normierungr;
    e_values_r2l[j]/=normierungr;
    e_values_r2r[j]/=normierungr;
  }
  /* Ausgeben nur der l2l-Werte fuer schnelles Reverb */
  printf("int e_delays[%d]={",early_tap_num);
  for (j=0;j<(early_tap_num-1);j++)
    printf("%d, ",e_delays_l2l[j]);
  printf("%d};\n\n",e_delays_l2l[j]);

  printf("float e_values[%d]={",early_tap_num);
  for (j=0;j<(early_tap_num-1);j++)
    printf("%0.4f, ",e_values_l2l[j]);
  printf("%0.4f};\n\n",e_values_l2l[j]);

}



/*
From: "Donald Schulz" <d.schulz@gmx.de>
To: <music-dsp@shoko.calarts.edu>
Subject: [music-dsp] Image-mirror method source code
Date: Sun, 11 Jun 2000 15:01:51 +0200

A while ago I wrote a program to calculate early echo responses.
As there seems to be some interest in this, I now post it into the
public domain.

Have phun,

Donald.
*/

/*

i have taken the liberty of renaming some of donald's variables to make
his code easier to use for english speaking, non-german speakers. i did
a simple search and replace on all the german words mentioned in the
note below. all of the comments are still in german, but the english
variable names make the code easier to follow, at least for me. i don't
think that i messed anything up by doing this, but if something's not
working you might want to try the original file, just in case.

douglas

*/

/*****************************************************************************
 *
 * Early Echo Computation using image-mirror method
```

```
 *
 * Position of listener, 2 sound-sources, room-size may be set.
 * Four early echo responses are calculated (from left sound source and
 * right sound source to left and right ear). Angle with which the sound
 * meets the ears is taken into account.
 * The early echo response from left sound source to left ear is printed
 * to screen for demonstration, the first table contains the delay times
 * and the second one the weights.
 *
 * Program is released into the public domain.
 *
 * Sorry for german comments :-(
 * Some frequently used german words:
 * hoerpos : listening position
 * breite : width
 * laenge : length
 * hoehe : height
 * daempfung : damping
 * links : left
 * rechts : right
 * Ohr    : ear
 * Winkel : angle
 * Wichtung : weight
 * Normierung : normalization
 *
 *
 * If someone does some improvements on this, I (Donald, d.schulz@gmx.de)
 * would be happy to get the improved code.
 *
 ***************************************************************************/



#include <math.h>
#include <stdio.h>

#define early_length 0x4000     /* Length der Puffer fuer early-echo */
#define early_length_1 0x3fff

#define early_tap_num 20        /* Anzahl an early-echo taps */

#define width 15.0              /* 15 m widthr Raum (x)*/
#define length 20.0             /* 20 m lang (y) */
#define height 10.0               /* 10 m hoch (z)*/
#define damping 0.999         /* Dampingsfaktor bei Reflexion */

#define listening_positionx 7.91            /* hier sitzt der Hoerer (linkes ear) */
#define listening_positiony 5.0
#define listening_positionz 2.0

#define leftposx 5.1            /* hier steht die linke Schallquelle */
#define leftposy 16.3
#define leftposz 2.5

#define rightposx 5.9            /* hier steht die rechte Schallquelle */
#define rightposy 6.3
#define rightposz 1.5

#define i_length 32             /* Length des Eingangs-Zwischenpuffers */
#define i_length_1 31
#define o_length 32             /* Length des Ausgangs-Zwischenpuffers */
#define o_length_1 31


float *early_l2l;  /* linker Kanal nach linkem ear */
float *early_l2r;  /* linker Kanal nach rechtem ear */
float *early_r2l;  /* rechter Kanal nach linkem ear */
float *early_r2r;  /* rechter Kanal nach rechtem ear */

int early_pos=0;

int e_delays_l2l[early_tap_num];       /* Delays der early-echos */
float e_values_l2l[early_tap_num];     /* Geweighten der delays */
int e_delays_l2r[early_tap_num];
float e_values_l2r[early_tap_num];
int e_delays_r2l[early_tap_num];
float e_values_r2l[early_tap_num];
int e_delays_r2r[early_tap_num];
float e_values_r2r[early_tap_num];


/* Early-echo Berechnung mittels Spiegelquellenverfahren
```

```
Raummodell:
H - Hoererposition
L - Spiegelschallquellen des linken Kanales
U - Koordinatenursprung
Raum sei 11 meter breit und 5 meter lang (1 Zeichen = 1 meter)
Linker Kanal stehe bei x=2  y=4  z=?
Hoerer stehe bei x=5 y=1 z=?
____________________________________________________________________
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
__L___L________________L___L________________L___L________________L_
L | L        |       L | L       |        L | L       |        L|
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |    H    |          |         |         |
_________________________X__________________________________________
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
|          |         |         |          |         |         |
__L___L________________L___L_________________L__L________________L_
*/

main()
{
  int i,j,select;
  float dist_max;
  float x,y,z,xref,yref,zref;
  float x_pos,y_pos,z_pos;
  float distance,angle;
  float weight;
  float normalizationr,normalizationl;


  early_l2l=(float *)malloc(early_length*sizeof(float));
  early_l2r=(float *)malloc(early_length*sizeof(float));;
  early_r2l=(float *)malloc(early_length*sizeof(float));;
  early_r2r=(float *)malloc(early_length*sizeof(float));;


  /* Erst mal Echos loeschen: */
  for (i=0;i<early_length;i++)
  early_l2l[i]=early_l2r[i]=early_r2l[i]=early_r2r[i]=0.0;

  dist_max=300.0*early_length/44100.0;  /* 300 m/s Schallgeschwindigkeit */

  /* Echo vom LINKEN Kanal auf linkes/rechtes ear berechnen */

  for (x=-ceil(dist_max/(2*length));x<=ceil(dist_max/(2*length));x++)
  for (y=-ceil(dist_max/(2*width));y<=ceil(dist_max/(2*width));y++)
  for (z=-ceil(dist_max/(2*height));z<=ceil(dist_max/(2*height));z++)
  {
    xref=2*x*width;
    yref=2*y*length;
    zref=2*z*height;
    for (select=0;select<8;select++)   /* vollstaendige Permutation */
    {
      if (select&1) x_pos=xref+leftposx;
      else x_pos=xref-leftposx;
      if (select&2) y_pos=yref+leftposy;
      else y_pos=yref-leftposy;
      if (select&4) z_pos=zref+leftposz;
      else z_pos=zref-leftposz;
      /* Jetzt steht die absolute Position der Quelle in ?_pos */

      /* Relative Position zum linken ear des Hoerers bestimmen: */
      x_pos-=listening_positionx;
      y_pos-=listening_positiony;
      z_pos-=listening_positionz;


      distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
      /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
      if ((distance*147)<early_length)
      {
        /* Einfallsangle im Bereich -Pi/2 bis Pi/2 ermitteln: */
        angle=atan(y_pos/x_pos);
        if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
        {  /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
          angle+=3.1415926/2;     weight = 1 - angle/4;
        }
```

```
      else            /* Klang kommt von hinten: */
      {
        angle-=3.1415926/2;  weight= 1 + angle/4;
      }
      /* Early-echo gemaess angle und Entfernung gewichten: */
      early_l2l[(int) (distance*147.)]+=weight/(pow(distance,3.1));
    }

    /* Relative Position zum rechten ear des Hoerers bestimmen: */
    x_pos-=0.18;  /* Kopf ist 18 cm breit */

    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallsangle im Bereich -Pi/2 bis Pi/2 ermitteln: */
      angle=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
        angle-=3.1415926/2;     weight = 1 + angle/4;
      }
      else            /* Klang kommt von hinten: */
      {
        angle+=3.1415926/2;  weight= 1 - angle/4;
      }
      /* Early-echo gemaess angle und Entfernung gewichten: */
      early_l2r[(int) (distance*147.)]+=weight/(pow(distance,3.1));
    }
  }
}

/* Echo vom RECHTEN Kanal auf linkes/rechtes ear berechnen */

for (x=-ceil(dist_max/(2*length));x<=ceil(dist_max/(2*length));x++)
for (y=-ceil(dist_max/(2*width));y<=ceil(dist_max/(2*width));y++)
for (z=-ceil(dist_max/(2*height));z<=ceil(dist_max/(2*height));z++)
{
  xref=2*x*width;
  yref=2*y*length;
  zref=2*z*height;
  for (select=0;select<8;select++)   /* vollstaendige Permutation */
  {
    if (select&1) x_pos=xref+rightposx;
    else x_pos=xref-rightposx;
    if (select&2) y_pos=yref+rightposy;
    else y_pos=yref-rightposy;
    if (select&4) z_pos=zref+rightposz;
    else z_pos=zref-rightposz;
    /* Jetzt steht die absolute Position der Quelle in ?_pos */

    /* Relative Position zum linken ear des Hoerers bestimmen: */
    x_pos-=listening_positionx;
    y_pos-=listening_positiony;
    z_pos-=listening_positionz;


    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147.)<early_length)
    {
      /* Einfallsangle im Bereich -Pi/2 bis Pi/2 ermitteln: */
      angle=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
        angle+=3.1415926/2;     weight = 1 - angle/4;
      }
      else            /* Klang kommt von hinten: */
      {
        angle-=3.1415926/2;  weight= 1 + angle/4;
      }
      /* Early-echo gemaess angle und Entfernung gewichten: */
      early_r2l[(int) (distance*147.)]+=weight/(pow(distance,3.1));
    }


    /* Und jetzt Early-Echo zweiter Kanal auf LINKES ear berechnen */
    x_pos-=0.18; /* Kopfwidth addieren */

    distance=sqrt(pow(x_pos,2)+pow(y_pos,2)+pow(z_pos,2));
    /* distance*147 = distance/300[m/s]*44100[ATW/s]; bei 32kHZ-> *106 */
    if ((distance*147)<early_length)
    {
      /* Einfallsangle im Bereich -Pi/2 bis Pi/2 ermitteln: */
```

```
      angle=atan(y_pos/x_pos);
      if (y_pos>0)   /* Klang kommt aus vorderem Bereich: */
      {  /* 0=left=>Verstaerkung=1 Pi=right=>Verstaerkung=0.22 (?) */
        angle-=3.1415926/2;     weight = 1 + angle/4;
      }
      else            /* Klang kommt von hinten: */
      {
        angle+=3.1415926/2;  weight= 1 - angle/4;
      }
      /* Early-echo gemaess angle und Entfernung gewichten: */
      early_r2r[(int) (distance*147.)]+=weight/(pow(distance,3.1));
    }
  }
}


/* Und jetzt aus berechnetem Echo die ersten early_tap_num Werte holen */
/* Erst mal e's zuruecksetzen: */
for (i=0;i<early_tap_num;i++)
{
  e_values_l2l[i]=e_values_l2r[i]=0.0;
  e_delays_l2l[i]=e_delays_l2r[i]=0;  /* Unangenehme Speicherzugriffe vermeiden */
  e_values_r2l[i]=e_values_r2r[i]=0.0;
  e_delays_r2l[i]=e_delays_r2r[i]=0;
}

/* und jetzt e_delays und e_values extrahieren: */
j=0;
normalizationl=normalizationr=0.0;
for(i=0;i<early_length;i++)
{
  if (early_l2l[i]!=0)
  {
    e_delays_l2l[j]=i;
    e_values_l2l[j]=early_l2l[i];
    normalizationl+=early_l2l[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_l2r[i]!=0)
  {
    e_delays_l2r[j]=i;
    e_values_l2r[j]=early_l2r[i];
    normalizationr+=early_l2r[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_r2l[i]!=0)
  {
    e_delays_r2l[j]=i;
    e_values_r2l[j]=early_r2l[i];
    normalizationl+=early_r2l[i];
    j++;
  }
  if (j==early_tap_num) break;
}
j=0;
for(i=0;i<early_length;i++)
{
  if (early_r2r[i]!=0)
  {
    e_delays_r2r[j]=i;
    e_values_r2r[j]=early_r2r[i];
    normalizationr+=early_r2r[i];
    j++;
  }
  if (j==early_tap_num) break;
}

/* groessere von beiden normalizationen verwenden: */
if (normalizationr>normalizationl) normalizationr=normalizationl;

for (j=0;j<early_tap_num;j++)
{
  e_values_l2l[j]/=normalizationr;
```

```
    e_values_l2r[j]/=normalizationr;
    e_values_r2l[j]/=normalizationr;
    e_values_r2r[j]/=normalizationr;
  }
  /* Ausgeben nur der l2l-Werte fuer schnelles Reverb */
  printf("int e_delays[%d]={",early_tap_num);
  for (j=0;j<(early_tap_num-1);j++)
    printf("%d, ",e_delays_l2l[j]);
  printf("%d};\n\n",e_delays_l2l[j]);

  printf("float e_values[%d]={",early_tap_num);
  for (j=0;j<(early_tap_num-1);j++)
    printf("%0.4f, ",e_values_l2l[j]);
  printf("%0.4f};\n\n",e_values_l2l[j]);

}
```

ECE320 project: Reverberation w/ parameter control from PC (click this to go back to the index)

References : Posted by Brahim Hamadicharef (project by Hua Zheng and Shobhit Jain)

Linked file : rev.txt (this linked file is included below)

Notes :
rev.asm
ECE320 project: Reverberation w/ parameter control from PC
Hua Zheng and Shobhit Jain
12/02/98 ~ 12/11/98
(se linked file)

Linked files

```
; rev.asm
;
; ECE320 project: Reverberation w/ parameter control from PC
;
; Hua Zheng and Shobhit Jain
; 12/02/98 ~ 12/11/98
;
; bugs fixed
; wrong co coefficients
; using current out point to calculate new in point
; r6 changed in set_dl (now changed to r4)
; initialize er delaylines to be 0 causes no output -- program memory
; periodic pops: getting garbage because external memory is configured to 16k

;================================================
; Initialization
;================================================

SAMPLERATE equ 48
    nolist
    include 'core302.asm'
    list

;--------------------------
; Variables and storage setup
;--------------------------

RESET  equ 255  ; serial data reset character
n_para equ 29  ; number of parameters expected from serial port
delayline_number equ 12
delay_pointers equ 24
er_number equ 3
co_number equ 6
ap_number equ 3

 org x:$0
ser_data ds n_para ; location of serial data chunk
delays   ; default rev parameters: delay length
 dc 967   ; er1 $3c7
 dc 1867   ; er2 $74b
 dc 3359   ; er3 $d1f
 dc 2399   ; co1 $95f
 dc 2687   ; co2 $a7f
 dc 2927   ; co3 $b6f
 dc 3271   ; co4 $cc7
 dc 3457   ; co5 $d81
 dc 3761   ; co6 $eb1
 dc 293   ; ap1 $125
 dc 83   ; ap2 $53
 dc 29   ; ap3 $1d
coeffs   ; default rev parameters: coefficients
 dc 0.78   ; er1
 dc 0.635   ; er2
 dc 0.267   ; er3
; dc 0
; dc 0
; dc 0
 dc 0.652149  ; co1 $7774de
 dc 0.301209
 dc 0.615737  ; co2 $53799e
 dc 0.334735
 dc 0.586396  ; co3 $4ed078
 dc 0.362044
 dc 0.546884  ; co4 $4b0f06
 dc 0.399249
 dc 0.525135  ; co5 $4337a0
 dc 0.419943
 dc 0.493653  ; co6 $3f3006
```

```
 dc 0.450179
 dc 0.2   ; brightness
 dc 0.4   ; mix
comb ds 6   ; one sample delay in comb filters
in ds 1   ; input sample
lpf  ds 1   ; one sample delay in LPF
dl_p ds delay_pointers ; delayline in/out pointers

dl_er_l equ $1000  ; max er delayline length 4096/48=85.3ms
dl_co_l equ $1000  ; max co delayline length 85.3ms
dl_ap_l equ $200  ; max ap delayline length 512/48=10.67ms

 org p:$1000
dl_er1 dsm dl_er_l ; er1 delayline
dl_er2 dsm dl_er_l ; er2 delayline
dl_er3 dsm dl_er_l ; er3 delayline
 org y:$8000
dl_co1 dsm dl_co_l ; co1 delayline
dl_co2 dsm dl_co_l ; co2 delayline
dl_co3 dsm dl_co_l ; co3 delayline
dl_co4 dsm dl_co_l ; co4 delayline
dl_co5 dsm dl_co_l ; co5 delayline
dl_co6 dsm dl_co_l ; co6 delayline
 org y:$F000
dl_ap1 dsm dl_ap_l ; ap1 delayline
dl_ap2 dsm dl_ap_l ; ap2 delayline
dl_ap3 dsm dl_ap_l ; ap3 delayline

;---------------------------
; Memory usage
;---------------------------
; P:$0000 -- $0200 core file
; P:$0200 -- $0300 progam
;
; X:$0000 -- $1BFF data 7168=149.3ms  serial data, parameters, pointers
; Y:$0000 -- $1BFF data 7168=149.3ms  not used
; P:$1000 -- $4FFF data 16384=341.3ms er(85*3=255ms)
; Y:$8000 -- $FFFF data 32768=682.67ms co(80*6=480ms) and ap(10*3=30ms)
;
; X,Y:$1C00 -- $1BFF reserved for system
;

;=================================================
; Main program
;=================================================

 org     p:$200
main

;---------------------------
; Initialization
;---------------------------

 move #0,x0
; move #dl_er1,r0
; move #dl_er_l,y0
; do #er_number,clear_dl_er_loop
; rep y0
;  movem x0,p:(r0)+
; nop
;clear_dl_er_loop

 move #dl_co1,r0
 move #dl_co_l,y0
 do #co_number,clear_dl_co_loop
 rep y0
  move x0,y:(r0)+
 nop
clear_dl_co_loop

 move #dl_ap1,r0
 move #dl_ap_l,y0
 do #ap_number,clear_dl_ap_loop
 rep y0
  move x0,y:(r0)+
 nop
clear_dl_ap_loop

 move #comb,r0
 rep #co_number
  move x0,x:(r0)+ ; init comb filter states
 move #lpf,r0
 move x0,x:(r0)  ; init LPF state
```

```
 move #ser_data,r6 ; incoming data buffer pointer
 move #(n_para-1),m6

 jsr set_dl  ; set all delayline pointers

 ; initialize SCI
 movep #$0302,x:M_SCR ; R/T enable, INT disable
 movep #$006a,x:M_SCCR   ; SCP=0, CD=106 (9636 bps)
 movep #7,x:M_PCRE

;--------------------------
; Main loop
; Register usage
;  r0: delayline pointer pointer
;  r1: coefficients pointer
;  r2: comb filter internal state pointer
;  r4,r5: used in delayline subroutine
;  r6: incoming buffer pointer
;  a: output of each segment
;--------------------------

main_loop:

 move #dl_p,r0
 move #coeffs,r1

 waitdata r3,buflen,1
 move x:(r3)+,a
 move a,x:<in  ; save input sample for mix

;--------------------------
; Early reflection
;--------------------------
; temp = in;
; for (int i=0; i<earlyRefNum; i++)
; {
;  in = delayLineEarlyRef[c][i]->tick(in);
;  temp += earlyRefCoeff[i] * in;
; }
; return temp;

 move a,b    ; b=temp=in
 move #(dl_er_l-1),n6
 do #er_number,er_loop
  jsr use_dl_er
  move a,y0   ; y0=a=in (delayline out)
  move x:(r1)+,x0  ; x0=coeff
  mac x0,y0,b  ; b=temp
er_loop
 asr #2,b,a
 move b,a

;--------------------------
; Comb filter
;--------------------------
; float temp1 = 0., temp2;
; for (int i=0; i<combNum; i++)
; {
;  temp2 = delayLineComb[c][i]->tick
;   (in + combCoeff[i] * combLastOut[c][i]);
;  combLastOut[c][i] = temp2+combDamp[i]*combLastOut[c][i];
;  temp1 += temp2;
; }
; return temp1 / float(combNum);

 move #comb,r2
 move a,y1
 clr b
 move #(dl_co_l-1),n6
 do #co_number,co_loop
  move y1,a   ; a=in
  move x:(r1)+,x0  ; x0=coeff
  move x:(r2),y0  ; y0=lastout
  mac x0,y0,a x:(r1)+,x0 ; x0=damp
  jsr use_dl
  move a,x1   ; a=x1=temp2
  mac x0,y0,a  ; a=lastout
  move a,x:(r2)+
  add x1,b   ; b=temp1
co_loop
; asr #2,b,a
 move b,a
```

```
;---------------------------
; All-pass filter
;---------------------------
; float temp1, temp2;
; for (int i=0; i<allPassNum; i++)
; {
;  temp1 = delayLineAllPass[c][i]->lastOut();
;  temp2 = allPassCoeff[i] * (in - temp1);
;  delayLineAllPass[c][i]->tick(in + temp2);
;  in = temp1 + temp2;
; }
; return in;

 move #1,n0
 move #0.7,x1
 move #(dl_ap_l-1),n6
 do #ap_number,ap_loop
  move y:(r0+n0),x0 ; x0=temp1
  sub x0,a   ; a=in-temp1
  move a,y0
  mpy x1,y0,b  ; b=temp2
  add b,a   ; a=in+temp2
  jsr use_dl
  add x0,b   ; b=temp1+temp2
  move b,a   ; a=in
ap_loop

;---------------------------
; Brightness
;---------------------------
; lastout = lastout + BW * (in - lastout);
 move x:<lpf,b
 sub b,a  x:(r1)+,x0 ; a=in-lastout, x0=bri
 move a,y0
 mpy x0,y0,a
 add b,a
 move a,x:<lpf

;---------------------------
; Mix
;---------------------------
; out = (1-mix)*in + mix*out = in + mix * (out - in);
 move x:<in,y0   ; y0=in
 sub y0,a  x:(r1)+,x0 ; a=out-in, x0=mix
 move a,y1    ; y1=out-in
 mpy x0,y1,b y0,a  ; b=mix*(out-in), a=in
 add b,a

;---------------------------
; Spit out
;---------------------------
 move a,y:(r3)+
 move a,y:(r3)+
 move (r3)+

;---------------------------
; Get parameters and reformat them
;---------------------------
 jclr #2,x:M_SSR,main_loop  ; do nothing if no new data arrived

 movep x:M_SRXL,a    ; get next 8-bit word from SCI

 cmp #RESET,a
 jeq reformat_data   ; if it's RESET, then reformat data

 move a,x:(r6)+    ; save one incoming data for later reformatting

 jmp main_loop

reformat_data:
; order of parameters:
; er1 delay, er1 coeff, er2 ..., er3 ...
; co1 delay, coeff_c, coeff_d, co2 ... , ... , co6
; ap1 delay, ap2, ap3
; brightness
; mix

 move #ser_data,r0
 move #delays,r1
 move #coeffs,r2

 do #3,format_er_loop
```

```
  move x:(r0)+,a  ; er delay
  asr #20,a,a
  ; max delay 4096=2^12, max value 256=2^8, scale 256/4096=2^-4
  move a0,x:(r1)+
  move x:(r0)+,a  ; er coeff
  asr #9,a,a
  move a0,x:(r2)+
format_er_loop

 move #>$000001,x0
 do #6,format_co_loop
  move x:(r0)+,a  ; co delay
  asr #20,a,a  ; max delay 4096=2^12
  move a0,a1
;try this:  asl #4,a,a
  or x0,a
  move a1,x:(r1)+
  move x:(r0)+,a  ; co coeff
  asr #9,a,a
  move a0,x:(r2)+
  move x:(r0)+,a  ; co damping
  asr #9,a,a
  move a0,x:(r2)+
format_co_loop

 do #3,format_ap_loop
  move x:(r0)+,a  ; ap delay
  asr #23,a,a  ; max delay 528=2^9
  move a0,a1
  or x0,a
  move a1,x:(r1)+
format_ap_loop

 jsr set_dl

 move x:(r0)+,a   ; brightness
 asr #9,a,a
 move a0,x:(r2)+

 move x:(r0)+,a   ; mix
 asr #9,a,a
 move a0,x:(r2)+

 jmp main_loop

;================================================
; Set all delayline length subroutine
; IN: nothing
; OUT: out pointer UNCHANGED
;  in pointer = out + length e.g. (#(dl_p+3))=(#(dl_p+4))+x:(r4)
;  r4=r4+1: next delay length
;================================================
set_dl:
 move #(dl_p+1),r5 ; first out pointer
 move #dl_er1,r4
 move r4,x:(r5)+  ; initial out point=delayline starting address
 move (r5)+
 move #dl_er2,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_er3,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_co1,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_co2,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_co3,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_co4,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_co5,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_co6,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_ap1,r4
 move r4,x:(r5)+
```

```
 move (r5)+
 move #dl_ap2,r4
 move r4,x:(r5)+
 move (r5)+
 move #dl_ap3,r4
 move r4,x:(r5)+
 move (r5)+

 move #delays,r4   ; delayline length
 move #(dl_p+1),r5  ; first out pointer
 move #2,n5
 do #delayline_number,set_dl_loop
  move x:(r4)+,x0  ; x0=length
  move x:(r5)-,a  ; a=out pointer
  add x0,a
  move a,x:(r5)+  ; in=out+length
  move (r5)+n5  ; next out pointer
set_dl_loop
 rts

;=================================================
; Access delayline subroutine
; IN: in and out pointers in r4,r5
;  modulo (delayline length-1) in n6
;  input sample in a
; OUT: in and out pointers modulo incremented
;  output sample in a
;=================================================
; inputs[inPoint++] = sample;
; inPoint &= lengthm1;
; lastOutput = inputs[outPoint++];
; outPoint &= lengthm1;
; return lastOutput;
use_dl:
 move n6,m4
 move n6,m5
 move x:(r0)+,r4 ; in point
 move x:(r0)-,r5 ; out point
 move a,y:(r4)+ ; queue in
 move y:(r5)+,a ; queue out
 move r4,x:(r0)+ ; modulo incremented in point
 move r5,x:(r0)+ ; modulo incremented out point
 rts

use_dl_er:   ; using P memory
 move n6,m4
 move n6,m5
 move x:(r0)+,r4 ; in point
 move x:(r0)-,r5 ; out point
 movem a,p:(r4)+ ; queue in
 movem p:(r5)+,a ; queue out
 move r4,x:(r0)+ ; modulusly incremented in point
 move r5,x:(r0)+ ; modulusly incremented out point
 rts
```

Guitar feedback (click this to go back to the index)

References : Posted by Sean Costello

Notes :
It is fairly simple to simulate guitar feedback with a simple Karplus-Strong algorithm (this was described in a CMJ article in the early 90's):

Code :
```
Run the output of the Karplus-Strong delay lines into a nonlinear shaping function for distortion (i.e. 6
parallel delay lines for 6 strings, going into 1 nonlinear shaping function that simulates an overdriven
amplifier, fuzzbox, etc.);

Run part of the output into a delay line, to simulate the distance from the amplifier to the "strings";

The delay line feeds back into the Karplus-Strong delay lines. By controlling the amount of the output fed
into the delay line, and the length of the delay line, you can control the intensity and pitch of the feedback
note.
```

Comments
from : ignatz@webmail.co.za
comment : any C snippet code ???

thx

Lo-Fi Crusher (click this to go back to the index)

Type : Quantizer / Decimator with smooth control

References : Posted by David Lowenfels

Notes :
Yet another bitcrusher algorithm. But this one has smooth parameter control.

Normfreq goes from 0 to 1.0; (freq/samplerate)
Input is assumed to be between 0 and 1.
Output gain is greater than unity when bits < 1.0;

Code :
```
function output = crusher( input, normfreq, bits );
    step = 1/2^(bits);
    phasor = 0;
    last = 0;

    for i = 1:length(input)
       phasor = phasor + normfreq;
       if (phasor >= 1.0)
          phasor = phasor - 1.0;
          last = step * floor( input(i)/step + 0.5 ); %quantize
       end
       output(i) = last; %sample and hold
    end
end
```

Most simple and smooth feedback delay (click this to go back to the index)

Type : Feedback delay

References : Posted by antiprosynthesis[AT]hotmail[DOT]com

Notes :
fDlyTime = delay time parameter (0-1)

i = input index
j = delay index

Code :
```
if( i >= SampleRate )
    i = 0;

j = i - (fDlyTime * SampleRate);

if( j < 0 )
    j += SampleRate;

Output = DlyBuffer[ i++ ] = Input + (DlyBuffer[ j ] * fFeedback);
```

Comments
from : antiprosynthesis@hotmail.com
comment : This algo didn't seem to work on testing again, just change:

Output = DlyBuffer[i++] = Input + (DlyBuffer[j] * fFeedback);
----
to

Output = DlyBuffer[i] = Input + (DlyBuffer[j] * fFeedback);

i++;
----
and it will work fine.

from : antiprosynthesis@hotmail.com
comment : Here's a more clear source. both BufferSize and MaxDlyTime are amounts of samples. BufferSize should best be 2*MaxDlyTime to have proper sound.
-
```
if( i >= BufferSize )
   i = 0;

j = i - (fDlyTime * MaxDlyTime);

if( j < 0 )
   j += BufferSize;

Output = DlyBuffer[ i ] = Input + (DlyBuffer[ j ] * fFeedback);

i++;
```

Most simple static delay (click this to go back to the index)

Type : Static delay

References : Posted by antiprosynthesis[AT]hotmail[DOT]com

Notes :
This is the most simple static delay (just delays the input sound an amount of samples). Very useful for newbies also probably very easy to change in a feedback delay (for comb filters for example).

Note: fDlyTime is the delay time parameter (0 to 1)

i = input index
j = output index

Code :
```
if( i >= SampleRate )
    i = 0;

DlyBuffer[ i ] = Input;

j = i - (fDlyTime * SampleRate);

i++;

if( j < 0 )
    j = SampleRate + j;

Output = DlyBuffer[ j ];
```

Comments
from : antiprosynthesis@hotmail.com
comment : Another note: The delay time will be 0 if fDlyTime is 0 or 1.

from : anonymous@fake.org
comment : I think you should be careful with mixing floats and integers that way (error-prone, slow float-to-int conversions, etc).

This should also work (haven't checked, not best way of doing it):

... (initializing) ..

```
float numSecondsDelay = 0.3f;
int numSamplesDelay_ = (int)(numSecondsDelay * sampleRate); // maybe want to round to an integer instead of truncating..

float *buffer_ = new float[2*numSamplesDelay];

for (int i = 0; i < numSamplesDelay_; ++i)
{
  buffer_[i] = 0.f;
}

int readPtr_ = 0;
int writePtr_ = numSamplesDelay_;
```

... (processing) ...

```
for (i = each sample)
{
  buffer_[writePtr_] = input[i];
  output[i] = buffer_[readPtr_];

  ++writePtr_;
  if (writePtr_ >= 2*numSamplesDelay_)
    writePtr_ = 0;

  ++readPtr_;
  if (readPtr_ >= 2*numSamplesDelay_)
    readPtr_ = 0;
}
```

Parallel combs delay calculation (click this to go back to the index)

References : Posted by Juhana Sadeharju (kouhia[AT]nic[DOT]funet[DOT]fi)

Notes :
This formula can be found from a patent related to parallel combs structure. The formula places the first echoes coming out of parallel combs to uniformly distributed sequence. If T_ ,...,T_n are the delay lines in increasing order, the formula can be derived by setting T_(k-1)/T_k = Constant and T_n/(2*T_1) = Constant, where 2*T_1 is the echo coming just after the echo T_n.
I figured this out myself as it is not told in the patent. The formula is not the best which one can come up. I use a search method to find echo sequences which are uniform enough for long enough time. The formula is uniform for a short time only.
The formula doesn't work good for series allpass and FDN structures, for which a similar formula can be derived with the same idea. The search method works for these structures as well.

Phaser code (click this to go back to the index)

References : Posted by Ross Bencina

Linked file : phaser.cpp (this linked file is included below)

Notes :
(see linked file)

Comments
from : dj_ikke@hotmail.com
comment : What range should be the float parameter of the Update function? From -1 to 1, 0 to 1 or -32768 to 32767?

from : rossb@audiomulch.com
comment : It doesn't matter what the range of the parameter to the Update function is. Usually in a floating point signal chain you would use -1 to 1,but anything else will work just as well.

from : askywhale@free.fr
comment : Please what is the usual range of frequencies ?

Linked files
```
/*
Date: Mon, 24 Aug 1998 07:02:40 -0700
Reply-To: music-dsp
Originator: music-dsp@shoko.calarts.edu
Sender: music-dsp
Precedence: bulk
From: "Ross Bencina" <rbencina@hotmail.com>
To: Multiple recipients of list <music-dsp>
Subject: Re: Phaser revisited [code included]
X-Comment:  Music Hackers Unite! http://shoko.calarts.edu/~glmrboy/musicdsp/music-dsp.html
Status: RO

Hi again,

Thanks to Chris Towsend and Marc Lindahl for their helpful
contributions. I now have a working phaser and it sounds great! It seems
my main error was using a 'sub-sampled' all-pass reverberator instead of
a single sample all-pass filter [what was I thinking? :)].

I have included a working prototype (C++) below for anyone who is
interested. My only remaining doubt is whether the conversion from
frequency to delay time [ _dmin = fMin / (SR/2.f); ] makes any sense
what-so-ever.

Ross B.
*/
/*
    class: Phaser
    implemented by: Ross Bencina <rossb@kagi.com>
    date: 24/8/98

    Phaser is a six stage phase shifter, intended to reproduce the
    sound of a traditional analogue phaser effect.
    This implementation uses six first order all-pass filters in
    series, with delay time modulated by a sinusoidal.

    This implementation was created to be clear, not efficient.
    Obvious modifications include using a table lookup for the lfo,
    not updating the filter delay times every sample, and not
    tuning all of the filters to the same delay time.

    Thanks to:
    The nice folks on the music-dsp mailing list, including...
    Chris Towsend and Marc Lindahl

    ...and Scott Lehman's Phase Shifting page at harmony central:
    http://www.harmony-central.com/Effects/Articles/Phase_Shifting/

*/

#define SR (44100.f)  //sample rate
#define F_PI (3.14159f)

class Phaser{
public:
 Phaser()  //initialise to some usefull defaults...
     : _fb( .7f )
     , _lfoPhase( 0.f )
     , _depth( 1.f )
```

```
        , _zm1( 0.f )
    {
     Range( 440.f, 1600.f );
     Rate( .5f );
    }

    void Range( float fMin, float fMax ){ // Hz
     _dmin = fMin / (SR/2.f);
        _dmax = fMax / (SR/2.f);
    }

    void Rate( float rate ){ // cps
     _lfoInc = 2.f * F_PI * (rate / SR);
    }

    void Feedback( float fb ){ // 0 -> <1.
     _fb = fb;
    }

    void Depth( float depth ){  // 0 -> 1.
      _depth = depth;
    }

    float Update( float inSamp ){
     //calculate and update phaser sweep lfo...
        float d  = _dmin + (_dmax-_dmin) * ((sin( _lfoPhase ) +
1.f)/2.f);
        _lfoPhase += _lfoInc;
        if( _lfoPhase >= F_PI * 2.f )
         _lfoPhase -= F_PI * 2.f;

        //update filter coeffs
        for( int i=0; i<6; i++ )
         _alps[i].Delay( d );

        //calculate output
        float y =  _alps[0].Update(
            _alps[1].Update(
                      _alps[2].Update(
                       _alps[3].Update(
                        _alps[4].Update(
                         _alps[5].Update( inSamp + _zm1 * _fb ))))));
        _zm1 = y;

        return inSamp + y * _depth;
    }
private:
 class AllpassDelay{
    public:
     AllpassDelay()
         : _a1( 0.f )
            , _zm1( 0.f )
            {}

        void Delay( float delay ){ //sample delay time
         _a1 = (1.f - delay) / (1.f + delay);
        }

        float Update( float inSamp ){
         float y = inSamp * -_a1 + _zm1;
         _zm1 = y * _a1 + inSamp;

            return y;
        }
    private:
     float _a1, _zm1;
    };

    AllpassDelay _alps[6];

    float _dmin, _dmax; //range
    float _fb; //feedback
    float _lfoPhase;
    float _lfoInc;
    float _depth;

    float _zm1;
};
```

Reverberation Algorithms in Matlab (click this to go back to the index)

References : Posted by Gautham J. Mysore (gauthamjm [AT] yahoo [DOT] com)

Linked file : MATLABReverb.zip

Notes :
These M-files implement a few reverberation algorithms (based on Schroeder's and Moorer's algorithms). Each of the M-files include a short description.

There are 5 M-files that implement reverberation. They are:

- schroeder1.m
- schroeder2.m
- schroeder3.m
- moorer.m
- stereoverb.m

The remaining 8 M-files implement filters, delay lines etc. Most of these are used in the above M-files. They can also be used as building blocks for other reverberation algorithms.

Comments
from : brewjinm@aol.com
comment : StereoVerb is the name of an old car stereo "enhancer" from way back. I was just trying to find it's roots.

Reverberation techniques (click this to go back to the index)

References : Posted by Sean Costello

Notes :
* Parallel comb filters, followed by series allpass filters. This was the original design by Schroeder, and was extended by Moorer. Has a VERY metallic sound for sharp transients.

* Several allpass filters in serie (also proposed by Schroeder). Also suffers from metallic sound.

* 2nd-order comb and allpass filters (described by Moorer). Not supposed to give much of an advantage over first order sections.

* Nested allpass filters, where an allpass filter will replace the delay line in another allpass filter. Pioneered by Gardner. Haven't heard the results.

* Strange allpass amp delay line based structure in Jon Dattorro article (JAES). Four allpass filters are used as an input to a cool "figure-8" feedback loop, where four allpass reverberators are used in series with
a few delay lines. Outputs derived from various taps in structure. Supposedly based on a Lexicon reverb design. Modulating delay lines are used in some of the allpass structures to "spread out" the eigentones.

* Feedback Delay Networks. Pioneered by Puckette/Stautner, with Jot conducting extensive recent research. Sound VERY good, based on initial experiments. Modulating delay lines and feedback matrixes used to spread out eigentones.

* Waveguide-based reverbs, where the reverb structure is based upon the junction of many waveguides. Julius Smith developed these. Recently, these have been shown to be essentially equivalent to the feedback delay network reverbs. Also sound very nice. Modulating delay lines and scattering values used to spread out eigentones.

* Convolution-based reverbs, where the sound to be reverbed is convolved with the impulse response of a room, or with exponentially-decaying white noise. Supposedly the best sound, but very computationally expensive, and not very flexible.

* FIR-based reverbs. Essentially the same as convolution. Probably not used, but shorter FIR filters are probably used in combination with many of the above techniques, to provide early reflections.

smsPitchScale Source Code (click this to go back to the index)

Type : Pitch Scaling (often incorrectly referred to as "Pitch Shifting") using the Fourier transform

References : Posted by sms[AT]dspdimension.com

Linked file : http://www.dspdimension.com

Code :
```
See above web site
```

Comments
  from : chinagreatman@163.net
  comment : Hi:
 Not Bad.

Soft saturation (click this to go back to the index)

Type : waveshaper

References : Posted by Bram de Jong

Notes :
This only works for positive values of x. a should be in the range 0..1

Code :
```
x < a:
  f(x) = x
x > a:
  f(x) = a + (x-a)/(1+((x-a)/(1-a))^2)
x > 1:
  f(x) = (a+1)/2
```

Time compression-expansion using standard phase vocoder (click this to go back to the index)

Type : vocoder phase time stretching

References : Posted by Cournape

Linked file : vocoder.m (this linked file is included below)

Notes :
Standard phase vocoder. For imporved techniques (faster), see paper of Laroche : "Improved phase vocoder time-scale modification of audio"
Laroche, J.; Dolson, M.
Speech and Audio Processing, IEEE Transactions on , Volume: 7 Issue: 3 , May 1999
Page(s): 323 -332

Comments
from : dsp[at]rymix.net
comment : Anyone know what language this is in? It really would be nice to understand the syntax.

from : null@null.null
comment : It's matlab code see mathworks.
/Daniel

from : dsp@rymix.net
comment : thanks =)

from : ericlee280.at.hotmail.com
comment : The code seems to contain an undefined variable lss_frame, can someone explain what this is?

from : yyc@cad.el.yuntech.edu.tw
comment : The code seems to contain an undefined variable lss_frame, can someone explain what this is?

from : cournape[at]enst[dot]fr
comment : There is indeed an error in the script. I will post the correction to the administrator.

For now, here is the correction. You have to replace
lss_frame by Ls (which is properly defined before the main loop in the script which is online). When I checked the code, there can be also some out of range error for the output vector : a change in the max variable definition seems to solve the problem (at least for overlapp below 0.75).

replace max = (nb_frame-2)*La+Nfft
by max = (nb_frame)*La+Nfft.

Linked files
```
function [S] = vocoder(E,Nfft,over,alpha)

% Phase vocoder time-scaling :

% - E     : input vector
% - Nfft  : size per frame
% - over  : overlapp ( between 0 et 1 )
% - alpha : time-expansion/compression factor ( alpha < 1 : compression; alpha > 1 : expansion ).
%
% 30.11.02. Cournapeau David.
%
% Time expansion using standart phase vocoder technique. Based on Dolson and Laroche's paper :
% "NEW PHASE-VOCODER TECHNIQUES FOR PITCH-SHIFTING, HARMONIZING AND
% OTHER EXOTIC EFFECTS", in


%===================================================================%
% The synthesis signal's length isn't exactly alpha*input's length. %
%===================================================================%


% Verifiy the overlapp

if( over < 0 | over >= 1 )
   error('error : overlapp must be between 0 and 1');
end;

if ( alpha <= 0)
    error('alpha must be strictly positive');
end;

E    = E(:);
N    = length(E);
Nfft = 2^(nextpow2(Nfft));
```

```
% Computing vocoder's parameters :
% - La        : number of samples to "advance" for each anamysis frame : analysis hop size.
% - nb_frames : number of frames to compute
% - Ls        : number of samples ot "advance" for each synthesis frame : synthesis hop size.
% - S         : S is the result vector
% - h         : hanning window

La        = floor((1-over) * Nfft);
nb_frames = floor((N-Nfft) / La);
max    = (nb_frames-2)*La + Nfft;
ls        = floor(alpha * La);

S         = zeros(floor(max*alpha),1);
h         = hanning(Nfft);

% Init process :

X     = h.*E(1:Nfft);
tX    = fft(X,Nfft);
Phis1 = angle(tX)
Phia1 = Phis1;

for loop=2:nb_frames-1

    %============================================================================
    % ( classic analysis part of a phase vocoder )

    % Take a frame, and windowing it

    X  = h.*E((loop-1) * La + 1:(loop-1)*La + Nfft);

    % XI is the amplitude spectrum, and Phia2 the phase spectrum.

    tX    = fft(X, Nfft);
 Xi    = abs(tX);
    Phia2 = angle(tX);

    %=============================================================================
    % the part which actually does the time scaling

    % One compute the actual pulsations, and shift them. The tricky part is here...

    omega = mod( (Phia2-Phia1)-2*pi*([0:Nfft-1].')/Nfft * La + pi, 2*pi) - pi;
    omega = 2 * pi * ([0:Nfft-1].') / Nfft + omega / La;
    Phis2 = Phis1 + lss_frame*omega;

    % The new phases values  :

    Phis1 = Phis2;
    Phia1 = Phia2;

    %===========================================================================
    %  Synthetise the frame, thanks to the computed phase and amplitude spectrum :
    % ( classic synthetisis part of a phase vocoder )

    tfs = Xi.*exp(j*Phis2);
    Xr  = real(ifft(tfs)).*h;

    % overlapp-add the synthetised frame Xr

    S((loop-1)*lss_frame+1:(loop-1)*lss_frame+Nfft) ...
      = S((loop-1)*lss_frame+1:(loop-1)*lss_frame+Nfft) + Xr;

end;
```

Variable-hardness clipping function (click this to go back to the index)

References : Posted by Laurent de Soras <laurent@ohmforce.com>

Linked file : laurent.gif

Notes :
k >= 1 is the "clipping hardness". 1 gives a smooth clipping, and a high value gives hardclipping.

Don't set k too high, because the formula use the pow() function, which use exp() and would overflow easily. 100 seems to be a reasonable value for "hardclipping"

Code :
```
f (x) = sign (x) * pow (atan (pow (abs (x), k)), (1 / k));
```

Comments
from : antiprosynthesis@hotmail.com
comment : Use this function instead of atan and see performance increase drastically :)

```
inline double fastatan( double x )
{
 return (x / (1.0 + 0.28 * (x * x)));
}
```

WaveShaper (click this to go back to the index)

Type : waveshaper

References : Posted by Bram de Jong

Notes :
where x (in [-1..1] will be distorted and a is a distortion parameter that goes from 1 to infinity
The equation is valid for positive and negativ values.
If a is 1, it results in a slight distortion and with bigger a's the signal get's more funky.

A good thing about the shaper is that feeding it with bigger-than-one
values, doesn't create strange fx. The maximum this function will reach is
1.2 for a=1.

Code :
```
f(x,a) = x*(abs(x) + a)/(x^2 + (a-1)*abs(x) + 1)
```

Waveshaper (click this to go back to the index)

Type : waveshaper

References : Posted by Jon Watte

Notes :
A favourite of mine is using a sin() function instead.
This will have the "unfortunate" side effect of removing
odd harmonics if you take it to the extreme: a triangle
wave gets mapped to a pure sine wave.
This will work with a going from .1 or so to a= 5 and bigger!
The mathematical limits for a = 0 actually turns it into a linear
function at that point, but unfortunately FPUs aren't that good
with calculus :-) Once a goes above 1, you start getting clipping
in addition to the "soft" wave shaping. It starts getting into
more of an effect and less of a mastering tool, though :-)

Seeing as this is just various forms of wave shaping, you
could do it all with a look-up table, too. In my version, that would
get rid of the somewhat-expensive sin() function.

Code :
```
(input: a == "overdrive amount")

z = M_PI * a;
s = 1/sin(z)
b = 1/a

if (x > b)
  f(x) = 1
else
  f(x) = sin(z*x)*s
```

Waveshaper (click this to go back to the index)

References : Posted by Partice Tarrabia and Bram de Jong

Notes :
amount should be in [-1..1[Plot it and stand back in astonishment! ;)

Code :
```
x = input in [-1..1]
y = output
k = 2*amount/(1-amount);

f(x) = (1+k)*x/(1+k*abs(x))
```

Comments
from : kaleja@estarcion.com
comment : I haven't compared this to the other waveshapers, but its behavior with input outside the [-1..1] range is interesting. With a relatively moderate shaping amounts which don't distort in-range signals severely, it damps extremely out-of-range signals fairly hard, e.g. x = 100, k = 0.1 yields y = 5.26; as x goes to infinity, y approaches 5.5. This might come in handy to control nonlinear processes which would otherwise be prone to computational blowup.

Waveshaper (simple description) (click this to go back to the index)

Type : Polynomial; Distortion

References : Posted by Jon Watte

Notes :
> The other question; what's a 'waveshaper' algorithm. Is it simply another
> word for distortion?

A typical "waveshaper" is some function which takes an input sample value X and transforms it to an output sample X'. A typical implementation would be a look-up table of some number of points, and some level of interpolation between those points (say, cubic). When people talk about a wave shaper, this is most often what they mean. Note that a wave shaper, as opposed to a filter, does not have any state. The mapping from X -> X' is stateless.

Some wave shapers are implemented as polynomials, or using other math functions. Hard clipping is a wave shaper implemented using the min() and max() functions (or the three-argument clamp() function, which is the same thing). A very mellow and musical-sounding distortion is implemented using a third-degree polynomial; something like X' = (3/2)X - (1/2)X^3. The nice thing with polynomial wave shapers is that you know that the maximum they will expand bandwidth is their order. Thus, you need to oversample 3x to make sure that a third-degree polynomial is aliasing free. With a lookup table based wave shaper, you don't know this (unless you treat an N-point table as an N-point polynomial :-)

Code :
```
float waveshape_distort( float in ) {
  return 1.5f * in - 0.5f * in *in * in;
}
```

Waveshaper :: Gloubi-boulga (click this to go back to the index)

References : Laurent de Soras on IRC

Notes :
Multiply input by gain before processing

Code :
```
const double x = input * 0.686306;
const double a = 1 + exp (sqrt (fabs (x)) * -0.75);
output = (exp (x) - exp (-x * a)) / (exp (x) + exp (-x));
```

16-to-8-bit first-order dither (click this to go back to the index)

Type : First order error feedforward dithering code

References : Posted by Jon Watte

Notes :
This is about as simple a dithering algorithm as you can implement, but it's likely to sound better than just truncating to N bits.

Note that you might not want to carry forward the full difference for infinity. It's probably likely that the worst performance hit comes from the saturation conditionals, which can be avoided with appropriate instructions on many DSPs and integer SIMD type instructions, or CMOV.

Last, if sound quality is paramount (such as when going from > 16 bits to 16 bits) you probably want to use a higher-order dither function found elsewhere on this site.

Code :
```
// This code will down-convert and dither a 16-bit signed short
// mono signal into an 8-bit unsigned char signal, using a first
// order forward-feeding error term dither.

#define uchar unsigned char

void dither_one_channel_16_to_8( short * input, uchar * output, int count, int * memory )
{
  int m = *memory;
  while( count-- > 0 ) {
    int i = *input++;
    i += m;
    int j = i + 32768 - 128;
    uchar o;
    if( j < 0 ) {
      o = 0;
    }
    else if( j > 65535 ) {
      o = 255;
    }
    else {
      o = (uchar)((j>>8)&0xff);
    }
    m = ((j-32768+128)-i);
    *output++ = o;
  }
  *memory = m;
}
```

3rd order Spline interpollation (click this to go back to the index)

References : Posted by Dave from Muon Software, originally from Josh Scholar

Notes :
(from Joshua Scholar about Spline interpollation in general...)
According to sampling theory, a perfect interpolation could be found by replacing each sample with a sinc function centered on that sample, ringing at your target nyquest frequency, and at each target point you just sum all of contributions from the sinc functions of every single point in source.
The sinc function has ringing that dies away very slowly, so each target sample will have to have contributions from a large neighborhood of source samples. Luckily, by definition the sinc function is bandwidth limited, so once we have a source that is prefilitered for our target nyquest frequency and reasonably oversampled relative to our nyquest frequency, ordinary interpolation techniques are quite fruitful even though they would be pretty useless if we hadn't oversampled.

We want an interpolation routine that at very least has the following characteristics:

1. Obviously it's continuous. But since finite differencing a signal (I don't really know about true differentiation) is equivalent to a low frequency attenuator that drops only about 6 dB per octave, continuity at the higher derivatives is important too.

2. It has to be stiff enough to find peaks when our oversampling missed them. This is where what I said about the combination the sinc function's limited bandwidth and oversampling making interpolation possible comes into play.

I've read some papers on splines, but most stuff on splines relates to graphics and uses a control point descriptions that is completely irrelevant to our sort of interpolation. In reading this stuff I quickly came to the conclusion that splines:

1. Are just piecewise functions made of polynomials designed to have some higher order continuity at the transition points.

2. Splines are highly arbitrary, because you can choose arbitrary derivatives (to any order) at each transition. Of course the more you specify the higher order the polynomials will be.

3. I already know enough about polynomials to construct any sort of spline. A polynomial through 'n' points with a derivative specified at 'm[1]' points and second derivatives specified at 'm[2]' points etc. will be a polynomial of the order n-1+m[1]+m[2]...

A way to construct third order splines (that admittedly doesn't help you construct higher order splines), is to linear interpolate between two parabolas. At each point (they are called knots) you have a parabola going through that point, the previous and the next point. Between each point you linearly interpolate between the polynomials for each point. This may help you imagine splines.

As a starting point I used a polynomial through 5 points for each knot and used MuPad (a free Mathematica like program) to derive a polynomial going through two points (knots) where at each point it has the same first two derivatives as a 4th order polynomial through the surrounding 5 points. My intuition was that basing it on polynomials through 3 points wouldn't be enough of a neighborhood to get good continuity. When I tested it, I found that not only did basing it on 5 point polynomials do much better than basing it on 3 point ones, but that 7 point ones did nearly as badly as 3 point ones. 5 points seems to be a sweet spot.

However, I could have set the derivatives to a nearly arbitrary values - basing the values on those of polynomials through the surrounding points was just a guess.

I've read that the math of sampling theory has different interpretation to the sinc function one where you could upsample by making a polynomial through every point at the same order as the number of points and this would give you the same answer as sinc function interpolation (but this only converges perfectly when there are an infinite number of points). Your head is probably spinning right now - the only point of mentioning that is to point out that perfect interpolation is exactly as stiff as a polynomial through the target points of the same order as the number of target points.

Code :
```
//interpolates between L0 and H0 taking the previous (L1) and next (H1)
points into account
inline float ThirdInterp(const float x,const float L1,const float L0,const
float H0,const float H1)
{
    return
    L0 +
    .5f*
    x*(H0-L1 +
       x*(H0 + L0*(-2) + L1 +
          x*( (H0 - L0)*9 + (L1 - H1)*3 +
             x*((L0 - H0)*15 + (H1 -  L1)*5 +
                x*((H0 - L0)*6 + (L1 - H1)*2 )))));
}
```

Comments
from : a@a.com
comment : What is x ?

from : mike_rawes@ku.oc.oohay
comment : The samples being interpolated represent the wave amplitude at a particular instant of time, T - an impulse train. So each sample is the amplitude at T=0,1,2,3 etc.

The purpose of interpolation is to determine the amplitude, a, for an arbitrary t, where t is any real number:

```
p1    p0  a  n0     n1
:     :   :  :      :
0-------1---t---2-------3------> T
        :   :
        :   :
        <-x->
```

x = t - T(p0)

-
myk

from : mike_rawes@ku.oc.oohay
comment : Dang! My nice diagram had its spacing stolen, and it now makes no sense!

p1, p0, n0, n1 are supposed to line up with 0,1,2,3 respectively. a is supposed to line up with the t. And finally, <-x-> spans between 1 and t.

-
myk

from : fcanessa@terra.cl
comment : 1.- What is 5f ?

2.- How I can test this procedure?.

Thank you

from : joshscholar_REMOVE_THIS@yahoo.com
comment : This is years later. but just in case anyone has the same problem as fcanessa... In C or C++ you can append an 'f' to a number to make it single precision, so .5f is the same as .5

5-point spline interpollation (click this to go back to the index)

Type : interpollation

References : Joshua Scholar, posted by David Waugh

Code :
```
//nMask = sizeofwavetable-1 where sizeofwavetable is a power of two.
double interpolate(double* wavetable, int nMask, double location)
{
    /* 5-point spline*/

    int nearest_sample = (int) location;
    double x = location - (double) nearest_sample;

    double p0=wavetable[(nearest_sample-2)&nMask];
    double p1=wavetable[(nearest_sample-1)&nMask];
    double p2=wavetable[nearest_sample];
    double p3=wavetable[(nearest_sample+1)&nMask];
    double p4=wavetable[(nearest_sample+2)&nMask];
    double p5=wavetable[(nearest_sample+3)&nMask];

    return p2 + 0.04166666666*x*((p3-p1)*16.0+(p0-p4)*2.0
    + x *((p3+p1)*16.0-p0-p2*30.0- p4
    + x *(p3*66.0-p2*70.0-p4*33.0+p1*39.0+ p5*7.0- p0*9.0
    + x *( p2*126.0-p3*124.0+p4*61.0-p1*64.0- p5*12.0+p0*13.0
    + x *((p3-p2)*50.0+(p1-p4)*25.0+(p5-p0)*5.0)))));
};
```

Comments
from : joshscholarREMOVETHIS@yahoo.com
comment : The code works much better if you oversample before interpolating. If you oversample enough (maybe 4 to 6 times oversampling) then the results are audiophile quality.

Allocating aligned memory (click this to go back to the index)

Type : memory allocation

References : Posted by Benno Senoner

Notes :
we waste up to align_size + sizeof(int) bytes when we alloc a memory area.
We store the aligned_ptr - unaligned_ptr delta in an int located before the aligned area.
This is needed for the free() routine since we need to free all the memory not only the aligned area.
You have to use aligned_free() to free the memory allocated with aligned_malloc() !

Code :
```
/* align_size has to be a power of two !! */
void *aligned_malloc(size_t size, size_t align_size) {

  char *ptr,*ptr2,*aligned_ptr;
  int align_mask = align_size - 1;

  ptr=(char *)malloc(size + align_size + sizeof(int));
  if(ptr==NULL) return(NULL);

  ptr2 = ptr + sizeof(int);
  aligned_ptr = ptr2 + (align_size - ((size_t)ptr2 & align_mask));


  ptr2 = aligned_ptr - sizeof(int);
  *((int *)ptr2)=(int)(aligned_ptr - ptr);

  return(aligned_ptr);
}

void aligned_free(void *ptr) {

  int *ptr2=(int *)ptr - 1;
  ptr -= *ptr2;
  free(ptr);
}
```

Antialiased Lines (click this to go back to the index)

Type : A slow, ugly, and unoptimized but short method to perform antialiased lines in a framebuffer

References : Posted by arguru@smartelectronix.com

Notes :
Simple code to perform antialiased lines in a 32-bit RGBA (1 byte/component) framebuffer.

pframebuffer <- unsigned char* to framebuffer bytes (important: Y flipped line order! [like in the way Win32 CreateDIBSection works...])

client_height=framebuffer height in lines
client_width=framebuffer width in pixels (not in bytes)

This doesnt perform any clip checl so it fails if coordinates are set out of bounds.

sorry for the engrish

Code :
```
//
// By Arguru
//
void PutTransPixel(int const x,int const y,UCHAR const r,UCHAR const g,UCHAR const b,UCHAR const a)
{
 unsigned char* ppix=pframebuffer+(x+(client_height-(y+1))*client_width)*4;
 ppix[0]=((a*b)+(255-a)*ppix[0])/256;
 ppix[1]=((a*g)+(255-a)*ppix[1])/256;
 ppix[2]=((a*r)+(255-a)*ppix[2])/256;
}

void LineAntialiased(int const x1,int const y1,int const x2,int const y2,UCHAR const r,UCHAR const g,UCHAR
const b)
{
 // some useful constants first
 double const dw=x2-x1;
 double const dh=y2-y1;
 double const slx=dh/dw;
 double const sly=dw/dh;

 // determine wichever raster scanning behaviour to use
 if(fabs(slx)<1.0)
 {
  // x scan
  int tx1=x1;
  int tx2=x2;
  double raster=y1;

  if(x1>x2)
  {
   tx1=x2;
   tx2=x1;
   raster=y2;
  }

  for(int x=tx1;x<=tx2;x++)
  {
   int const ri=int(raster);

   double const in_y0=1.0-(raster-ri);
   double const in_y1=1.0-(ri+1-raster);

   PutTransPixel(x,ri+0,r,g,b,in_y0*255.0);
   PutTransPixel(x,ri+1,r,g,b,in_y1*255.0);

   raster+=slx;
  }
 }
 else
 {
  // y scan
  int ty1=y1;
  int ty2=y2;
  double raster=x1;

  if(y1>y2)
  {
   ty1=y2;
   ty2=y1;
   raster=x2;
  }

  for(int y=ty1;y<=ty2;y++)
  {
   int const ri=int(raster);

   double const in_x0=1.0-(raster-ri);
   double const in_x1=1.0-(ri+1-raster);
```

```
   PutTransPixel(ri+0,y,r,g,b,in_x0*255.0);
   PutTransPixel(ri+1,y,r,g,b,in_x1*255.0);

   raster+=sly;
  }
 }
}
```

Comments
  from : Gog
  comment : Sorry, but what does this have to do with music DSP ??

Base-2 exp (click this to go back to the index)

References : Posted by Laurent de Soras

Notes :
Linear approx. between 2 integer values of val. Uses 32-bit integers. Not very efficient but fastest than exp()
This code was designed for x86 (little endian), but could be adapted for big endian processors.
Laurent thinks you just have to change the (*(1 + (int *) &ret)) expressions and replace it by (*(int *) &ret). However, He didn't test it.

Code :
```
inline double fast_exp2 (const double val)
{
   int    e;
   double ret;

   if (val >= 0)
   {
      e = int (val);
      ret = val - (e - 1);
      ((*(1 + (int *) &ret)) &= ~(2047 << 20)) += (e + 1023) << 20;
   }
   else
   {
      e = int (val + 1023);
      ret = val - (e - 1024);
      ((*(1 + (int *) &ret)) &= ~(2047 << 20)) += e << 20;
   }
   return (ret);
}
```

Comments
from : subatomic@vrsource.org
comment :
Here is the code to detect little endian processor:

```
    union
    {
       short   val;
       char    ch[sizeof( short )];
    } un;
    un.val = 256; // 0x10;

    if (un.ch[1] == 1)
    {
       // then we're little
    }
```

I've tested the fast_exp2() on both little and big endian (intel, AMD, and motorola) processors, and the comment is correct.

Here is the completed function that works on all endian systems:

```
inline double fast_exp2( const double val )
{
   // is the machine little endian?
   union
   {
      short   val;
      char    ch[sizeof( short )];
   } un;
   un.val = 256; // 0x10;
   // if un.ch[1] == 1 then we're little

   // return 2 to the power of val (exp base2)
   int    e;
   double ret;

   if (val >= 0)
   {
      e = int (val);
      ret = val - (e - 1);

      if (un.ch[1] == 1)
         ((*(1 + (int *) &ret)) &= ~(2047 << 20)) += (e + 1023) << 20;
      else
         ((*((int *) &ret)) &= ~(2047 << 20)) += (e + 1023) << 20;
   }
   else
   {
      e = int (val + 1023);
      ret = val - (e - 1024);

      if (un.ch[1] == 1)
         ((*(1 + (int *) &ret)) &= ~(2047 << 20)) += e << 20;
      else
         ((*((int *) &ret)) &= ~(2047 << 20)) += e << 20;
```

```
    }

    return ret;
}
```

Block/Loop Benchmarking (click this to go back to the index)

Type : Benchmarking Tool

References : Posted by arguru[AT]smartelectronix[DOT]com

Notes :
Requires CPU with RDTSC support

Code :
```
// Block-Process Benchmarking Code using rdtsc
// useful for measure DSP block stuff
// (based on Intel papers)
// 64-bit precission
// VeryUglyCode(tm) by Arguru

// globals
UINT time,time_low,time_high;

// call this just before enter your loop or whatever
void bpb_start()
{
 // read time stamp to EAX
 __asm rdtsc;
 __asm mov time_low,eax;
 __asm mov time_high,edx;
}

// call the following function just after your loop
// returns average cycles wasted per sample
UINT bpb_finish(UINT const num_samples)
{
 __asm rdtsc
 __asm sub eax,time_low;
 __asm sub edx,time_high;
 __asm div num_samples;
 __asm mov time,eax;
 return time;
}
```

Calculate notes (java) (click this to go back to the index)

Type : Java class for calculating notes with different in params

References : Posted by larsby[AT]elak[DOT]org

Linked file : Frequency.java (this linked file is included below)

Notes :
Converts between string notes and frequencies and back. I vaguely remember writing bits of it, and I got it off the net somwhere so dont ask me

- Larsby

Linked files
```
public class Frequency extends Number
{

    private static final double PITCH_OF_A4 = 57D;
    private static final double FACTOR = 12D / Math.log(2D);
    private static final String NOTE_SYMBOL[] = {
        "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A",
        "A#", "B"
    };
    public static float frequencyOfA4 = 440F;
    private float frequency;

    public static final double getPitch(float f)
    {
        return getPitch(f);
    }

    public static final double getPitch(double d)
    {
        return 57D + FACTOR * Math.log(d / (double)frequencyOfA4);
    }

    public static final float getFrequency(double d)
    {
        return (float)(Math.exp((d - 57D) / FACTOR) * (double)frequencyOfA4);
    }

    public static final String makeNoteSymbol(double d)
    {
        int i = (int)(d + 120.5D);
        StringBuffer stringbuffer = new StringBuffer(NOTE_SYMBOL[i % 12]);
        stringbuffer.append(Integer.toString(i / 12 - 10));
        return new String(stringbuffer);
    }

    public static float valueOf(String s)
        throws IllegalArgumentException
    {
        try
        {
            return (new Float(s)).floatValue();
        }
        catch(NumberFormatException _ex) { }
        try
        {
            return getFrequency(parseNoteSymbol(s));
        }
        catch(IllegalArgumentException _ex)
        {
            throw new IllegalArgumentException("Neither a floating point number nor a valid note
symbol.");
        }
    }

    public static final int parseNoteSymbol(String s)
        throws IllegalArgumentException
    {
        s = s.trim().toUpperCase();
        for(int i = NOTE_SYMBOL.length - 1; i >= 0; i--)
        {
            if(!s.startsWith(NOTE_SYMBOL[i]))
                continue;
            try
            {
                return i + 12 * Integer.parseInt(s.substring(NOTE_SYMBOL[i].length()).trim());
            }
            catch(NumberFormatException _ex) { }
            break;
```

```
        }

        throw new IllegalArgumentException("not valid note symbol.");
    }

    public static void transformPitch(TextComponent textcomponent, boolean flag)
    {
        boolean flag1 = false;
        String s = textcomponent.getText();
        if(flag)
        {
            try
            {
                textcomponent.setText(Integer.toString((int)(getFrequency(parseNoteSymbol(s)) +
0.5F)));
                return;
            }
            catch(IllegalArgumentException _ex)
            {
                flag1 = true;
            }
            return;
        }
        try
        {
            textcomponent.setText(makeNoteSymbol(getPitch((new Float(s)).floatValue())));
            return;
        }
        catch(NumberFormatException _ex)
        {
            flag1 = true;
        }
    }

    public Frequency(float f)
    {
        frequency = 1.0F;
        frequency = f;
    }

    public Frequency(String s)
        throws IllegalArgumentException
    {
        frequency = 1.0F;
        frequency = valueOf(s);
    }

    public byte byteValue()
    {
        return (byte)(int)(frequency + 0.5F);
    }

    public short shortValue()
    {
        return (short)(int)(frequency + 0.5F);
    }

    public long longValue()
    {
        return (long)(frequency + 0.5F);
    }

    public int intValue()
    {
        return (int)(frequency + 0.5F);
    }

    public float floatValue()
    {
        return frequency;
    }

    public double doubleValue()
    {
        return (double)frequency;
    }

    public String toString()
    {
        return Integer.toString(intValue());
    }

    public String toNoteSymbol()
```

```
    {
        return makeNoteSymbol(getPitch(frequency));
    }

    public static void main(String[] args)
    {
        System.out.println(Frequency.parseNoteSymbol("C2"));
 System.out.println(Frequency.getFrequency(24));
    }
}
```

Center separation in a stereo mixdown (click this to go back to the index)

References : Posted by Thiburce BELAVENTURE

Notes :
One year ago, i found a little trick to isolate or remove the center in a stereo mixdown.

My method use the time-frequency representation (FFT). I use a min fuction between left and right channels (for each bin) to create the pseudo center. I apply a phase correction, and i substract this signal to the left and right signals.

Then, we can remix them after treatments (or without) to produce a stereo signal in output.

This algorithm (I called it "TBIsolator") is not perfect, but the result is very nice, better than the phase technic (L substract R...). I know that it is not mathematically correct, but as an estimation of the center, the exact match is very hard to obtain. So, it is not so bad (just listen the result and see).

My implementation use a 4096 FFT size, with overlap-add method (factor 2). With a lower FFT size, the sound will be more dirty, and with a 16384 FFT size, the center will have too much high frequency (I don't explore why this thing appears).

I just post the TBIsolator code (see FFTReal in this site for implement the FFT engine).

pIns and pOuts buffers use the representation of the FFTReal class (0 to N/2-1: real parts, N/2 to N-1: imaginary parts).

Have fun with the TBIsolator algorithm ! I hope you enjoy it and if you enhance it, contact me (it's my baby...).

P.S.: the following function is not optimized.

Code :
```
/* ============================================================= */
/* nFFTSize must be a power of 2                                 */
/* ============================================================= */
/* Usage examples:                                               */
/* - suppress the center:  fAmpL = 1.f, fAmpC = 0.f, fAmpR = 1.f */
/* - keep only the center: fAmpL = 0.f, fAmpC = 1.f, fAmpR = 0.f */
/* ============================================================= */

void processTBIsolator(float *pIns[2], float *pOuts[2], long nFFTSize, float fAmpL, float fAmpC, float fAmpR)
{
 float fModL, fModR;
 float fRealL, fRealC, fRealR;
 float fImagL, fImagC, fImagR;
 double u;

 for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++ )
 {
  fModL = pIns[0][i] * pIns[0][i] + pIns[0][j] * pIns[0][j];
  fModR = pIns[1][i] * pIns[1][i] + pIns[1][j] * pIns[1][j];

  // min on complex numbers
  if ( fModL > fModR )
  {
   fRealC = fRealR;
   fImagC = fImagR;
  }
  else
  {
   fRealC = fRealL;
   fImagC = fImagL;
  }

  // phase correction...
  u = fabs(atan2(pIns[0][j], pIns[0][i]) - atan2(pIns[1][j], pIns[1][i])) / 3.141592653589;

  if ( u >= 1 ) u -= 1.;

  u = pow(1 - u*u*u, 24);

  fRealC *= (float) u;
  fImagC *= (float) u;

  // center extraction...
  fRealL = pIns[0][i] - fRealC;
  fImagL = pIns[0][j] - fImagC;

  fRealR = pIns[1][i] - fRealC;
  fImagR = pIns[1][j] - fImagC;

  // You can do some treatments here...

  pOuts[0][i] = fRealL * fAmpL + fRealC * fAmpC;
  pOuts[0][j] = fImagL * fAmpL + fImagC * fAmpC;

  pOuts[1][i] = fRealR * fAmpR + fRealC * fAmpC;
  pOuts[1][j] = fImagR * fAmpR + fImagC * fAmpC;
 }
}
```

Comments

from : fmanchec@thiburce.com
comment : I am sorry, my source code is not totally correct.

1 - the for is:

```
for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++, j++ )
```

2 - the correct min is:

```
if ( fModL > fModR )
{
   fRealC = pIns[1][i];
   fImagC = pIns[1][j];
}
else
{
   fRealC = pIns[0][i];
   fImagC = pIns[0][j];
}
```

3 - in the phase correction:

```
if ( u >= 1 ) u -= 1.;
```

must be replaced by:

```
if ( u >= 1 ) u = 2 - u;
```

Thiburce 'TB' BELAVENTURE

Center separation in a stereo mixdown (click this to go back to the index)

References : Posted by Thiburce BELAVENTURE

Notes :
One year ago, i found a little trick to isolate or remove the center in a stereo mixdown.

My method use the time-frequency representation (FFT). I use a min fuction between left and right channels (for each bin) to create the pseudo center. I apply a phase correction, and i substract this signal to the left and right signals.

Then, we can remix them after treatments (or without) to produce a stereo signal in output.

This algorithm (I called it "TBIsolator") is not perfect, but the result is very nice, better than the phase technic (L substract R...). I know that it is not mathematically correct, but as an estimation of the center, the exact match is very hard to obtain. So, it is not so bad (just listen the result and see).

My implementation use a 4096 FFT size, with overlap-add method (factor 2). With a lower FFT size, the sound will be more dirty, and with a 16384 FFT size, the center will have too much high frequency (I don't explore why this thing appears).

I just post the TBIsolator code (see FFTReal in this site for implement the FFT engine).

pIns and pOuts buffers use the representation of the FFTReal class (0 to N/2-1: real parts, N/2 to N-1: imaginary parts).

Have fun with the TBIsolator algorithm ! I hope you enjoy it and if you enhance it, contact me (it's my baby...).

P.S.: the following function is not optimized.

Code :
```
/* ============================================================= */
/* nFFTSize must be a power of 2                                 */
/* ============================================================= */
/* Usage examples:                                               */
/* - suppress the center:  fAmpL = 1.f, fAmpC = 0.f, fAmpR = 1.f */
/* - keep only the center: fAmpL = 0.f, fAmpC = 1.f, fAmpR = 0.f */
/* ============================================================= */

void processTBIsolator(float *pIns[2], float *pOuts[2], long nFFTSize, float fAmpL, float fAmpC, float fAmpR)
{
 float fModL, fModR;
 float fRealL, fRealC, fRealR;
 float fImagL, fImagC, fImagR;
 double u;

 for ( long i = 0, j = nFFTSize / 2; i < nFFTSize / 2; i++ )
 {
  fModL = pIns[0][i] * pIns[0][i] + pIns[0][j] * pIns[0][j];
  fModR = pIns[1][i] * pIns[1][i] + pIns[1][j] * pIns[1][j];

  // min on complex numbers
  if ( fModL > fModR )
  {
   fRealC = fRealR;
   fImagC = fImagR;
  }
  else
  {
   fRealC = fRealL;
   fImagC = fImagL;
  }

  // phase correction...
  u = fabs(atan2(pIns[0][j], pIns[0][i]) - atan2(pIns[1][j], pIns[1][i])) / 3.141592653589;

  if ( u >= 1 ) u -= 1.;

  u = pow(1 - u*u*u, 24);

  fRealC *= (float) u;
  fImagC *= (float) u;

  // center extraction...
  fRealL = pIns[0][i] - fRealC;
  fImagL = pIns[0][j] - fImagC;

  fRealR = pIns[1][i] - fRealC;
  fImagR = pIns[1][j] - fImagC;

  // You can do some treatments here...

  pOuts[0][i] = fRealL * fAmpL + fRealC * fAmpC;
  pOuts[0][j] = fImagL * fAmpL + fImagC * fAmpC;

  pOuts[1][i] = fRealR * fAmpR + fRealC * fAmpC;
  pOuts[1][j] = fImagR * fAmpR + fImagC * fAmpC;
 }
}
```

Clipping without branching (click this to go back to the index)

Type : Min, max and clip

References : Posted by Laurent de Soras <laurent@ohmforce.com>

Notes :
It may reduce accuracy for small numbers. I.e. if you clip to [-1; 1], fractional part of the result will be quantized to 23 bits (or more, depending on the bit depth of the temporary results). Thus, 1e-20 will be rounded to 0. The other (positive) side effect is the denormal number elimination.

Code :
```
float max (float x, float a)
{
   x -= a;
   x += fabs (x);
   x *= 0.5;
   x += a;
   return (x);
}

float min (float x, float b)
{
   x = b - x;
   x += fabs (x)
   x *= 0.5;
   x = b - x;
   return (x);
}

float clip (float x, float a, float b)
{
   x1 = fabs (x-a);
   x2 = fabs (x-b);
   x = x1 + (a+b);
   x -= x2;
   x *= 0.5;
   return (x);
}
```

Comments
from : kleps@refx.net
comment : AFAIK, the fabs() is using if()...

from : andy[AT]vellocet.com
comment : fabs/fabsf do not use if and are quicker than:
if (x<0) x = -x;
Do the speed tests yourself if you don't believe me!

from : kaleja@estarcion.com
comment : Depends on CPU and optimization options, but yes, Visual C++/x86/full optimization uses intrinsic fabs, which is very cool.

from : lennart.denninger[AT]guerrilla-games.com
comment : And ofcourse you could always use one of those nifty bit-tricks for fabs :)

(Handy when you don't want to link with the math-library, like when coding a softsynth for a 4Kb-executable demo :))

from : andy@a2hd.com
comment : according to my benchmarks (using the cpu clock cycle counter), fabs and the 'nifty bit tricks' have identicle performance characterstics, EXCEPT that with the nifty bit trick, sometimes it has a -horrible- penalty, which depends on the context..., maybe it does not optimize consistently? I use libmath fabs now. (i'm using gcc-3.3/linux on a P3)

Constant-time exponent of 2 detector (click this to go back to the index)

References : Posted by Brent Lehman (mailbjl[AT]yahoo.com)

Notes :
In your common FFT program, you want to make sure that the frame you're working with has a size that is a power of 2. This tells you in just a few operations. Granted, you won't be using this algorithm inside a loop, so the savings aren't that great, but every little hack helps ;)

Code :
```
// Quit if size isn't a power of 2
if ((-size ^ size) & size) return;

// If size is an unsigned int, the above might not compile.
// You'd want to use this instead:
if (((~size + 1) ^ size) & size) return;
```

Comments
from : functorx@yahoo.com
comment : I think I prefer:

if (! (size & (size - 1))) return;

I'm not positive this is fewer instructions than the above, but I think it's easier to see why it works (n and n-1 will share bits unless n is a power of two), and it doesn't require two's-complement.

- Tom 7

Conversions on a PowerPC (click this to go back to the index)

Type : motorola ASM conversions

References : Posted by James McCartney

Code :

```
double ftod(float x) { return (double)x;
00000000: 4E800020  blr
    // blr == return from subroutine, i.e. this function is a noop

float dtof(double x) { return (float)x;
00000000: FC200818  frsp       fp1,fp1
00000004: 4E800020  blr

int ftoi(float x) { return (int)x;
00000000: FC00081E  fctiwz     fp0,fp1
00000004: D801FFF0  stfd       fp0,-16(SP)
00000008: 8061FFF4  lwz        r3,-12(SP)
0000000C: 4E800020  blr

int dtoi(double x) { return (int)x;
00000000: FC00081E  fctiwz     fp0,fp1
00000004: D801FFF0  stfd       fp0,-16(SP)
00000008: 8061FFF4  lwz        r3,-12(SP)
0000000C: 4E800020  blr

double itod(int x) { return (double)x;
00000000: C8220000  lfd        fp1,@1558(RTOC)
00000004: 6C608000  xoris      r0,r3,$8000
00000008: 9001FFF4  stw        r0,-12(SP)
0000000C: 3C004330  lis        r0,17200
00000010: 9001FFF0  stw        r0,-16(SP)
00000014: C801FFF0  lfd        fp0,-16(SP)
00000018: FC200828  fsub       fp1,fp0,fp1
0000001C: 4E800020  blr

float itof(int x) { return (float)x;
00000000: C8220000  lfd        fp1,@1558(RTOC)
00000004: 6C608000  xoris      r0,r3,$8000
00000008: 9001FFF4  stw        r0,-12(SP)
0000000C: 3C004330  lis        r0,17200
00000010: 9001FFF0  stw        r0,-16(SP)
00000014: C801FFF0  lfd        fp0,-16(SP)
00000018: EC200828  fsubs      fp1,fp0,fp1
0000001C: 4E800020  blr
```

Copy-protection schemes (click this to go back to the index)

References : Posted by Moyer, Andy

Notes :
This post of Andy sums up everything there is to know about copy-protection schemes:

"Build a great product and release improvements regularly so that people will be willing to spend the money on it, thus causing anything that is cracked to be outdated quickly. Build a strong relationship with your customers, because if they've already paid for one of your products, and were satisfied, chances are, they will be more likely to buy another one of your products. Make your copy protection good enough so that somebody can't just do a search in Google and enter in a published serial number, but don't make registered users jump through flaming hoops to be able to use the product. Also use various approaches to copy protection within a release, and vary those approaches over multiple releases so that a hacker that cracked your app's version 1.0 can't just run a recorded macro in a text editor to crack your version 2.0 software [this being simplified]."

Comments
from : ultrano@mail.bg
comment : Won't it be good to make several versions of 1.0 with the functions places being scrambled, and unused static data being changed? And if the product uses plugins, to make each plugin detect if the code has been changed?

Cubic interpollation (click this to go back to the index)

Type : interpollation

References : Posted by Olli Niemitalo

Linked file : other001.gif

Notes :
(see linkfile)
finpos is the fractional, inpos the integer part.

Code :
```
xm1 = x [inpos - 1];
x0  = x [inpos + 0];
x1  = x [inpos + 1];
x2  = x [inpos + 2];
a = (3 * (x0-x1) - xm1 + x2) / 2;
b = 2*x1 + xm1 - (5*x0 + x2) / 2;
c = (x1 - xm1) / 2;
y [outpos] = (((a * finpos) + b) * finpos + c) * finpos + x0;
```

Denormal DOUBLE variables, macro (click this to go back to the index)

References : Posted by Jon Watte

Notes :
Use this macro if you want to find denormal numbers and you're using doubles...

Code :
```
#if PLATFORM_IS_BIG_ENDIAN
#define INDEX 0
#else
#define INDEX 1
#endif
inline bool is_denormal( double const & d ) {
  assert( sizeof( d ) == 2*sizeof( int ) );
  int l = ((int *)&d)[INDEX];
  return (l&0x7fe00000) != 0;
}
```

Denormal numbers (click this to go back to the index)

References : Compiled by Merlijn Blaauw

Linked file : other001.txt (this linked file is included below)

Notes :
this text describes some ways to avoid denormalisation. Denormalisation happens when FPU's go mad processing very small numbers

Comments
from : andy@a2hd.com
comment : See also the entry about 'branchless min, max and clip' by Laurent Soras in this section,

Using the following function,

```
float clip (float x, float a, float b)
{
  x1 = fabs (x-a);
  x2 = fabs (x-b);
  x = x1 + (a+b);
  x -= x2;
  x *= 0.5;
  return (x);
}
```

If you apply clipping from -1.0 to 1.0 will have a side effect of squashing denormal numbers to zero due to loss of precision on the order of ~1.0.e-20. The upside is that it is branchless, but possibly more expensive than adding noise and certainly more so than adding a DC offset.

Linked files

```
Denormal numbers
----------------

Here's a small recap on all proposed solutions to prevent the FPU from
denormalizing:
When you feed the FPU really small values (what's the exact 'threshold'
value?) the CPU will go into denormal mode to maintain precision; as such
precision isn't required for audio applications and denormal operations are
MUCH slower than normal operations, we want to avoid them.

All methods have been proposed by people other than me, and things I say
here may be inaccurate or completely wrong :). 'Algorithms' have not been
tested and can be implemented more efficiently no doubt.
If I made some horrible mistakes, or left some stuff out, please let me/the
list know.


** Checking denormal processing solution:

To detect if a denormal number has occured, just trace your code and
look up on the STAT FPU register ... if 0x0002 flag is set then a denormal
operation occured (this flag stays fixed until next FINIT)

** Checking denormal macro solution:

NOTES:

This is the least computationally efficient method of the lot, but has the
advantage of being inaudible.
Please note that in every feedback loop, you should also check for
denormals (rendering it useless on algorithms with loads of filters,
feedback loops, etc).

CODE:

#define IS_DENORMAL(f) (((*(unsigned int *)&f)&0x7f800000)==0)

// inner-loop:

is1 = *(++in1); is1 = IS_DENORMAL(is1) ? 0.f : is1;
is2 = *(++in2); is2 = IS_DENORMAL(is2) ? 0.f : is2;

** Adding noise solution:

NOTES:

Less overhead than the first solution, but still 2 mem accesses. Because a
number of the values of denormalBuf will be denormals themselves, there
will always be *some* denormal overhead. However, a small percentage
denormals probably isn't a problem.
```

Use this eq. to calculate the appropriate value of id (presuming rand() generates evenly distrubuted values):
id = 1/percentageOfDenormalsAllowed * denormalThreshold
(I do not know the exact value of the denormalThreshold, the value at which the FPU starts to denormal).

Possible additions to this algorithm include, noiseshaping the noise buffer, which would allow a smaller value of percentageOfDenormalsAllowed without becomming audible - however, in some algorithms, with filters and such, I think this might cause the noise to be removed, thus rendering it useless. Checking for denormals on noise generation might have a similar effect I suspect.

CODE:

```
// on construction:
float **denormalBuf = new float[getBlockSize()];

float id = 1.0E-20;

for (int i = 0; i < getBlockSize(); i++)
{
   denormalBuf[i] = (float)rand()/32768.f * id;
}

// inner-loop:

float noise = *(++noiseBuf);
is1 = *(++in1) + noise;
is2 = *(++in2) + noise;
..
```

** Flipping number solution:

NOTES:

In my opinion the way-to-go method for most applications; very little overhead (no compare/if/etc or memory access needed), there isn't a percentage of the values that will denormal and it will be inaudible in most cases.
The exact value of id will probably differ from algorithm to algorithm, but the proposed value of 1.0E-30 seemed too small to me.

CODE:

```
// on construction:
float id = 1.0E-25;

// inner-loop:
is1 = *(++in1) + id;
is2 = *(++in2) + id;
id = -id;
..
```

** Adding offset solution:

NOTES:

This is the most efficient method of the lot and is also inaudible. However, some filters will probably remove the added DC offset, thus rendering it useless.

CODE:

```
// inner-loop:

is1 = *(++in1) + 1.0E-25;
is2 = *(++in2) + 1.0E-25;
```

** Fix-up solution

You can also walk through your filter and clamp any numbers which are close enough to being denormal at the end of each block, as long as your blocks are not too large. For instance, if you implement EQ using a bi-quad, you can check the delay slots at the end of each process() call, and if any slot has a magnitude of 10^-15 or smaller, you just clamp it to 0. This will ensure that your filter doesn't run for long times with denormal numbers; ideally (depending on the coefficients) it won't reach 10^-35 from the 10^-15 initial state within the time of one block of samples.
That solution uses the least cycles, and also has the nice property of generating absolute-0 output values for long stretches of absolute-0 input values; the others don't.

Denormal numbers, the meta-text (click this to go back to the index)

References : Laurent de Soras

Linked file : denormal.pdf

Notes :
This very interesting paper, written by Laurent de Soras (www.ohmforce.com) has everything you ever wanted to know about denormal numbers! And it obviously descibes how you can get rid of them too!

(see linked file)

Dither code (click this to go back to the index)

Type : Dither with noise-shaping

References : Posted by Paul Kellett

Notes :
This is a simple implementation of highpass triangular-PDF dither (a good general-purpose dither) with optional 2nd-order noise shaping (which lowers the noise floor by 11dB below 0.1 Fs).
The code assumes input data is in the range +1 to -1 and doesn't check for overloads!

To save time when generating dither for multiple channels you can re-use lower bits of a previous random number instead of calling rand() again. e.g. r3=(r1 & 0x7F)<<8;

Code :
```
  int   r1, r2;                //rectangular-PDF random numbers
  float s1, s2;                //error feedback buffers
  float s = 0.5f;              //set to 0.0f for no noise shaping
  float w = pow(2.0,bits-1);   //word length (usually bits=16)
  float wi= 1.0f/w;
  float d = wi / RAND_MAX;     //dither amplitude (2 lsb)
  float o = wi * 0.5f;         //remove dc offset
  float in, tmp;
  int   out;

//for each sample...

  r2=r1;                               //can make HP-TRI dither by
  r1=rand();                           //subtracting previous rand()

  in += s * (s1 + s1 - s2);            //error feedback
  tmp = in + o + d * (float)(r1 - r2); //dc offset and dither

  out = (int)(w * tmp);                //truncate downwards
  if(tmp<0.0f) out--;                  //this is faster than floor()

  s2 = s1;
  s1 = in - wi * (float)out;           //error
```

Dithering (click this to go back to the index)

References : Paul Kellett

Linked file : nsdither.txt (this linked file is included below)

Notes :
(see linked file)

```
Linked files
Noise shaped dither  (March 2000)
-------------------

This is a simple implementation of highpass triangular-PDF dither with
2nd-order noise shaping, for use when truncating floating point audio
data to fixed point.

The noise shaping lowers the noise floor by 11dB below 5kHz (@ 44100Hz
sample rate) compared to triangular-PDF dither. The code below assumes
input data is in the range +1 to -1 and doesn't check for overloads!

To save time when generating dither for multiple channels you can do
things like this:  r3=(r1 & 0x7F)<<8; instead of calling rand() again.



  int   r1, r2;                //rectangular-PDF random numbers
  float s1, s2;                //error feedback buffers
  float s = 0.5f;              //set to 0.0f for no noise shaping
  float w = pow(2.0,bits-1);   //word length (usually bits=16)
  float wi= 1.0f/w;
  float d = wi / RAND_MAX;     //dither amplitude (2 lsb)
  float o = wi * 0.5f;         //remove dc offset
  float in, tmp;
  int   out;


//for each sample...

  r2=r1;                               //can make HP-TRI dither by
  r1=rand();                           //subtracting previous rand()

  in += s * (s1 + s1 - s2);            //error feedback
  tmp = in + o + d * (float)(r1 - r2); //dc offset and dither

  out = (int)(w * tmp);                //truncate downwards
  if(tmp<0.0f) out--;                  //this is faster than floor()

  s2 = s1;
  s1 = in - wi * (float)out;           //error



--
paul.kellett@maxim.abel.co.uk
http://www.maxim.abel.co.uk
```

Double to Int (click this to go back to the index)

Type : pointer cast (round to zero, or 'trunctate')

References : Posted by many people, implementation by Andy M00cho

Notes :
-Platform independant, literally. You have IEEE FP numbers, this will work, as long as your not expecting a signed integer back larger than 16bits :)
-Will only work correctly for FP numbers within the range of [-32768.0,32767.0]
-The FPU must be in Double-Precision mode

Code :
```
typedef double lreal;
typedef float  real;
typedef unsigned long uint32;
typedef long int32;

   //2^36 * 1.5, (52-_shiftamt=36) uses limited precision to floor
   //16.16 fixed point representation

const lreal _double2fixmagic = 68719476736.0*1.5;
const int32 _shiftamt        = 16;

#if BigEndian_
        #define iexp_                           0
        #define iman_                           1
#else
        #define iexp_                           1
        #define iman_                           0
#endif //BigEndian_

// Real2Int
inline int32 Real2Int(lreal val)
{
   val= val + _double2fixmagic;
   return ((int32*)&val)[iman_] >> _shiftamt;
}

// Real2Int
inline int32 Real2Int(real val)
{
   return Real2Int ((lreal)val);
}

For the x86 assembler freaks here's the assembler equivalent:

__double2fixmagic    dd 000000000h,042380000h

fld    AFloatingPoint Number
fadd   QWORD PTR __double2fixmagic
fstp   TEMP
movsx  eax,TEMP+2
```

Envelope Follower (click this to go back to the index)

References : Posted by ers

Code :
```
#define V_ENVELOPE_FOLLOWER_NUM_POINTS 2000
class vEnvelopeFollower :
{
      public:
  vEnvelopeFollower();
  virtual ~vEnvelopeFollower();
  inline void Calculate(float *b)
  {
   envelopeVal -= *buff;
   if (*b < 0)
    envelopeVal += *buff = -*b;
   else
    envelopeVal += *buff = *b;
   if (buff++ == bufferEnd)
    buff = buffer;
  }
  void SetBufferSize(float value);
  void GetControlValue(){return envelopeVal / (float)bufferSize;}

 private:
  float buffer[V_ENVELOPE_FOLLOWER_NUM_POINTS];
  float *bufferEnd, *buff, envelopeVal;
  int bufferSize;
      float val;
};

vEnvelopeFollower::vEnvelopeFollower()
{
 bufferEnd = buffer + V_ENVELOPE_FOLLOWER_NUM_POINTS-1;
 buff = buffer;
 val = 0;
 float *b = buffer;
 do
 {
  *b++ = 0;
 }while (b <= bufferEnd);
 bufferSize = V_ENVELOPE_FOLLOWER_NUM_POINTS;
 envelopeVal= 0;
}

vEnvelopeFollower::~vEnvelopeFollower()
{

}

void vEnvelopeFollower::SetBufferSize(float value)
{

 bufferEnd = buffer + (bufferSize = 100 + (int)(value * ((float)V_ENVELOPE_FOLLOWER_NUM_POINTS-102)));
 buff = buffer;
 float val =  envelopeVal / bufferSize;
 do
 {
  *buff++ = val;
 }while (buff <= bufferEnd);
 buff = buffer;
}
```

Exponential parameter mapping (click this to go back to the index)

References : Posted by Russell Borogove

Notes :
Use this if you want to do an exponential map of a parameter (mParam) to a range (mMin - mMax).
Output is in mData...

Code :
```
float logmax = log10f( mMax );
float logmin = log10f( mMin );
float logdata = (mParam * (logmax-logmin)) + logmin;

mData = powf( 10.0f, logdata );
if (mData < mMin)
{
  mData = mMin;
}
if (mData > mMax)
{
  mData = mMax;
}
```

Comments
from : rerdavies@msn.com
comment : No point in using heavy functions when lighter-weight functions work just as well. Use ln instead of log10f, and exp instead of pow(10,x). Log-linear is the same, no matter which base you're using, and base e is way more efficient than base 10.

from : kaleja@estarcion.com
comment : Thanks for the tip. A set of VST param wrapper classes which offers linear float, exponential float, integer selection, and text selection controls, using this technique for the exponential response, can be found in the VST source code archive -- finally.

from : act_ion@yahoo.com
comment : Just made my day!
pretty useful :) cheers Aktion

fast abs/neg/sign for 32bit floats (click this to go back to the index)

Type : floating point functions

References : Posted by tobybear[AT]web[DOT]de

Notes :
Haven't seen this elsewhere, probably because it is too obvious? Anyway, these functions are intended for 32-bit floating point numbers only and should work a bit faster than the regular ones.

fastabs() gives you the absolute value of a float
fastneg() gives you the negative number (faster than multiplying with -1)
fastsgn() gives back +1 for 0 or positive numbers, -1 for negative numbers

Comments are welcome (tobybear[AT]web[DOT]de)

Cheers

Toby (www.tobybear.de)

Code :
```
// C/C++ code:
float fastabs(float f)
{int i=((*(int*)&f)&0x7fffffff);return (*(float*)&i);}

float fastneg(float f)
{int i=((*(int*)&f)^0x80000000);return (*(float*)&i);}

int fastsgn(float f)
{return 1+(((*(int*)&f)>>31)<<1);}

//Delphi/Pascal code:
function fastabs(f:single):single;
begin i:=longint((@f)^) and $7FFFFFFF;result:=single((@i)^) end;

function fastneg(f:single):single;
begin i:=longint((@f)^) xor $80000000;result:=single((@i)^) end;

function fastsgn(f:single):longint;
begin result:=1+((longint((@f)^) shr 31)shl 1) end;
```

Comments
from : tobybear@web.de
comment : Matthias (bekkah[AT]web[DOT]de) wrote me a mail with the following further improvements for the C++ parts of the code:

```
// C++ code:
inline float fastabs(const float f)
{int i=((*(int*)&f)&0x7fffffff);return (*(float*)&i);}

inline float fastneg(const float f)
{int i=((*(int*)&f)^0x80000000);return (*(float*)&i);}

inline int fastsgn(const float f)
{return 1+(((*(int*)&f)>>31)<<1);}
```

Thanks!

from : picoder@mail.ru
comment : Too bad these 'tricks' need two additional FWAITs to work in a raw FPU code. Maybe standard fabs and fneg are better? Although, that fastsgn() could be useful since there's no FPU equivalent for it.

Cheers,
Aleksey.

from : picoder@mail.ru
comment : I meant 'fchs' in place of 'fneg'.

from : david@brannvall.net
comment : I don't know if this is any faster, but atleast you can avoid some typecasting.

```
function fastabs(f: Single): Single; var i: Integer absolute f;
begin i := i and $7fffffff; Result := f; end;
```

from : chris@m-audio.com
comment : Note that a reasonable compiler should be able to perform these optimizations for you. I seem to recall that GCC in particular has the capability to replace calls to [f]abs() with instructions optimized for the platform.

Fast binary log approximations (click this to go back to the index)

Type : C code

References : Posted by musicdsp.org[AT]mindcontrol.org

Notes :
This code uses IEEE 32-bit floating point representation knowledge to quickly compute approximations to the log2 of a value. Both functions return under-estimates of the actual value, although the second flavour is less of an under-estimate than the first (and might be sufficient for using in, say, a dBV/FS level meter).

Running the test program, here's the output:

0.1: -4 -3.400000
1:   0 0.000000
2:   1 1.000000
5:   2 2.250000
100: 6 6.562500

Code :
```
// Fast logarithm (2-based) approximation
// by Jon Watte

#include <assert.h>

int floorOfLn2( float f ) {
  assert( f > 0. );
  assert( sizeof(f) == sizeof(int) );
  assert( sizeof(f) == 4 );
  return (((*(int *)&f)&0x7f800000)>>23)-0x7f;
}

float approxLn2( float f ) {
  assert( f > 0. );
  assert( sizeof(f) == sizeof(int) );
  assert( sizeof(f) == 4 );
  int i = (*(int *)&f);
  return (((i&0x7f800000)>>23)-0x7f)+(i&0x007fffff)/(float)0x800000;
}

// Here's a test program:

#include <stdio.h>

// insert code from above here

int
main()
{
  printf( "0.1: %d  %f\n", floorOfLn2( 0.1 ), approxLn2( 0.1 ) );
  printf( "1:   %d  %f\n", floorOfLn2( 1. ), approxLn2( 1. ) );
  printf( "2:   %d  %f\n", floorOfLn2( 2. ), approxLn2( 2. ) );
  printf( "5:   %d  %f\n", floorOfLn2( 5. ), approxLn2( 5. ) );
  printf( "100: %d  %f\n", floorOfLn2( 100. ), approxLn2( 100. ) );
  return 0;
}
```

Comments
  from : tobybear@web.de
  comment : Here is some code to do this in Delphi/Pascal:

```
function approxLn2(f:single):single;
begin
 result:=(((longint((@f)^) and $7f800000) shr 23)-$7f)+(longint((@f)^) and $007fffff)/$800000;
end;

function floorOfLn2(f:single):longint;
begin
 result:=(((longint((@f)^) and $7f800000) shr 23)-$7f);
end;
```

Cheers,

Tobybear
www.tobybear.de

Fast exp2 approximation (click this to go back to the index)

References : Posted by Laurent de Soras <laurent[AT]ohmforce[DOT]com>

Notes :
Partial approximation of exp2 in fixed-point arithmetic. It is exactly :
[0 ; 1[-> [0.5 ; 1[
f : x |-> 2^(x-1)
To get the full exp2 function, you have to separate the integer and fractionnal part of the number. The integer part may be processed by bitshifting. Process the fractionnal part with the function, and multiply the two results.
Maximum error is only 0.3 % which is pretty good for two mul ! You get also the continuity of the first derivate.

-- Laurent

Code :
```
// val is a 16-bit fixed-point value in 0x0 - 0xFFFF ([0 ; 1[)
// Returns a 32-bit fixed-point value in 0x80000000 - 0xFFFFFFFF ([0.5 ; 1[)
unsigned int fast_partial_exp2 (int val)
{
   unsigned int   result;

   __asm
   {
      mov eax, val
      shl eax, 15           ; eax = input [31/31 bits]
      or  eax, 080000000h   ; eax = input + 1  [32/31 bits]
      mul eax
      mov eax, edx          ; eax = (input + 1) ^ 2 [32/30 bits]
      mov edx, 2863311531   ; 2/3 [32/32 bits], rounded to +oo
      mul edx               ; eax = 2/3 (input + 1) ^ 2 [32/30 bits]
      add edx, 1431655766   ; + 4/3 [32/30 bits] + 1
      mov result, edx
   }

   return (result);
}
```

Fast log2 (click this to go back to the index)

References : Posted by Laurent de Soras

Code :
```
inline float fast_log2 (float val)
{
   assert (val > 0);

   int * const  exp_ptr = reinterpret_cast <int *> (&val);
   int          x = *exp_ptr;
   const int    log_2 = ((x >> 23) & 255) - 128;
   x &= ~(255 << 23);
   x += 127 << 23;
   *exp_ptr = x;

   return (val + log_2);
}
```

Comments
from : tobybear@web.de
comment : And here is some native Delphi/Pascal code that
does the same thing:

```
function fast_log2(val:single):single;
var log2,x:longint;
begin
 x:=longint((@val)^);
 log2:=((x shr 23) and 255)-128;
 x:=x and (not(255 shl 23));
 x:=x+127 shl 23;
 result:=single((@x)^)+log2;
end;
```

Cheers

Toby

www.tobybear.de

from : henry@cordylus.de
comment : instead of using this pointer casting expressions one can also use a enum like this:

enum FloatInt
{
float f;

from : henry@cordylus.de
comment : instead of using this pointer casting expressions one can also use a enum like this:

```
enum FloatInt
{
float f;
int l;
} p;
```

and then access the data with:

```
p.f = x;
p.l >>= 23;
```

Greetings, Henry

from : henry@cordylus.de
comment : Sorry :

didnt mean enum, ment UNION !!!

fast power and root estimates for 32bit floats (click this to go back to the index)

Type : floating point functions

References : Posted by tobybear[AT]web[DOT]de

Notes :
Original code by Stefan Stenzel (also in this archive, see "pow(x,4) approximation") - extended for more flexibility.

fastpow(f,n) gives a rather *rough* estimate of a float number f to the power of an integer number n (y=f^n). It is fast but result can be quite a bit off, since we directly mess with the floating point exponent.-> use it only for getting rough estimates of the values and where precision is not that important.

fastroot(f,n) gives the n-th root of f. Same thing concerning precision applies here.

Cheers

Toby (www.tobybear.de)

Code :
```
//C/C++ source code:
float fastpower(float f,int n)
{
 long *lp,l;
 lp=(long*)(&f);
 l=*lp;l-=0x3F800000l;l<<=(n-1);l+=0x3F800000l;
 *lp=l;
 return f;
}

float fastroot(float f,int n)
{
 long *lp,l;
 lp=(long*)(&f);
 l=*lp;l-=0x3F800000l;l>>=(n-1);l+=0x3F800000l;
 *lp=l;
 return f;
}

//Delphi/Pascal source code:
function fastpower(i:single;n:integer):single;
var l:longint;
begin
 l:=longint((@i)^);
 l:=l-$3F800000;l:=l shl (n-1);l:=l+$3F800000;
 result:=single((@l)^);
end;

function fastroot(i:single;n:integer):single;
var l:longint;
begin
 l:=longint((@i)^);
 l:=l-$3F800000;l:=l shr (n-1);l:=l+$3F800000;
 result:=single((@l)^);
end;
```

Float to int (click this to go back to the index)

References : Posted by Ross Bencina

Notes :
intel only

Code :
```
int truncate(float flt)
{
  int i;
  static const double half = 0.5f;
  _asm
  {
     fld flt
     fsub half
     fistp i
  }
  return i
}
```

Comments
from : jaha@smartelectronix.com
comment : Note: Round nearest doesn't work, because the Intel FPU uses Even-Odd rounding in order to conform to the IEEE floating-point standard: when the FPU is set to use the round-nearest rule, values whose fractional part is exactly 0.5
round toward the nearest *even* integer. Thus, 1.5 rounds to 2, 2.5 rounds to 2, 3.5 rounds to 4. This is quite disastrous for the FLOOR/CEIL functions if you use the strategy you describe.

from : sugonaut@yahoo.com
comment : This version below seems to be more accurate on my Win32/P4. Doesn't deal with the Intel FPU issue. A faster solution than c-style casts though. But you're not always going to get the most accurate conversion. Like the previous comment; 2.5 will convert to 2, but 2.501 will convert to 3.

```
int truncate(float flt)
{
   int i;
   _asm
   {
      fld flt
      fistp i
   }
   return i
}
```

Float-to-int, coverting an array of floats (click this to go back to the index)

References : Posted by Stefan Stenzel

Notes :
intel only

Code :
```
void f2short(float *fptr,short *iptr,int n)
{
_asm {
    mov     ebx,n
    mov     esi,fptr
    mov     edi,iptr
    lea     ebx,[esi+ebx*4]   ; ptr after last
    mov     edx,0x80008000    ; damn endianess confuses...
    mov     ecx,0x4b004b00    ; damn endianess confuses...
    mov     eax,[ebx]         ; get last value
    push    eax
    mov     eax,0x4b014B01
    mov     [ebx],eax              ; mark end
    mov     ax,[esi+2]
    jmp     startf2slp

;   Pad with nops to make loop start at address divisible
;   by 16 + 2, e.g. 0x01408062, don't ask why, but this
;   gives best performance. Unfortumately "align 16" does
;   not seem to work with my VC.
;   below I noted the measured execution times for different
;   nop-paddings on my Pentium Pro, 100 conversions.
;   saturation:  off pos neg


   nop         ;355 546 563 <- seems to be best
;   nop         ;951 547 544
;   nop         ;444 646 643
;   nop         ;444 646 643
;   nop         ;944 951 950
;   nop         ;358 447 644
;   nop         ;358 447 643
;   nop         ;358 544 643
;   nop         ;543 447 643
;   nop         ;643 447 643
;   nop         ;1047 546 746
;   nop         ;545 954 1253
;   nop         ;545 547 661
;   nop         ;544 547 746
;   nop         ;444 947 1147
;   nop         ;444 548 545
in_range:
    mov     eax,[esi]
    xor     eax,edx
saturate:
    lea     esi,[esi+4]
    mov     [edi],ax
    mov     ax,[esi+2]
    add     edi,2
startf2slp:
    cmp     ax,cx
    je      in_range
    mov     eax,edx
    js      saturate     ; saturate neg -> 0x8000
    dec     eax          ; saturate pos -> 0x7FFF
    cmp     esi,ebx      ; end reached ?
    jb      saturate
    pop     eax
    mov     [ebx],eax    ; restore end flag
    }
}
```

Comments
  from : magnus[at]smartelectronix.com
  comment :
_asm {
mov ebx,n
mov esi,fptr
mov edi,iptr
lea ebx,[esi+ebx*4] ; ptr after last
mov edx,0x80008000 ; damn endianess confuses...
mov ecx,0x4b004b00 ; damn endianess confuses...
mov eax,[ebx] ; get last value

I think the last line here reads outside the buffer.

Gaussian dithering (click this to go back to the index)

Type : Dithering

References : Posted by Aleksey Vaneev (picoder[AT]mail[DOT]ru)

Notes :
It is a more sophisticated dithering than simple RND. It gives the most low noise floor for the whole spectrum even without noise-shaping. You can use as big N as you can afford (it will not hurt), but 4 or 5 is generally enough.

Code :
```
Basically, next value is calculated this way (for RND going from -0.5 to 0.5):

dither = (RND+RND+...+RND) / N.
           \           /
            \_________/
              N times

If your RND goes from 0 to 1, then this code is applicable:

dither = (RND+RND+...+RND - 0.5*N) / N.
```

Gaussian random numbers (click this to go back to the index)

Type : random number generation

References : Posted by tobybear[AT]web[DOT]de

Notes :
// Gaussian random numbers
// This algorithm (adapted from "Natur als fraktale Grafik" by
// Reinhard Scholl) implements a generation method for gaussian
// distributed random numbers with mean=0 and variance=1
// (standard gaussian distribution) mapped to the range of
// -1 to +1 with the maximum at 0.
// For only positive results you might abs() the return value.
// The q variable defines the precision, with q=15 the smallest
// distance between two numbers will be 1/(2^q div 3)=1/10922
// which usually gives good results.

// Note: the random() function used is the standard random
// function from Delphi/Pascal that produces *linear*
// distributed numbers from 0 to parameter-1, the equivalent
// C function is probably rand().

Code :
```
const q=15;
      c1=(1 shl q)-1;
      c2=(c1 div 3)+1;
      c3=1/c1;

function GRandom:single;
begin
 result:=(2*(random(c2)+random(c2)+random(c2))-3*(c2-1))*c3;
end;
```

Hermite Interpolator (x86 ASM) (click this to go back to the index)

Type : Hermite interpolator in x86 assembly (for MS VC++)

References : Posted by robert.bielik@rbcaudio.com

Notes :
An "assemblified" variant of Laurent de Soras hermite interpolator. I tried to do calculations as parallell as I could muster, but there is almost certainly room for improvements. Right now, it works about 5.3 times (!) faster, not bad to start with...

Parameter explanation:
frac_pos: fractional value [0.0f - 1.0f] to interpolator
pntr: pointer to float array where:
  pntr[0] = previous sample (idx = -1)
  pntr[1] = current sample (idx = 0)
  pntr[2] = next sample (idx = +1)
  pntr[3] = after next sample (idx = +2)

The interpolation takes place between pntr[1] and pntr[2].

Regards,
/Robert Bielik
RBC Audio

Code :
```
const float c_half = 0.5f;

__declspec(naked) float __hermite(float frac_pos, const float* pntr)
{
 __asm
 {
  push ecx;
  mov  ecx, dword ptr[esp + 12];
//////////////////////////////////////////////////////////////////////////////////////////////////
  add  ecx, 0x04;   // ST(0)  ST(1)  ST(2)  ST(3)  ST(4)  ST(5)  ST(6)  ST(7)
  fld  dword ptr [ecx+4]; // x1
  fsub dword ptr [ecx-4]; // x1-xm1
  fld  dword ptr [ecx]; // x0   x1-xm1
  fsub dword ptr [ecx+4]; // v   x1-xm1
  fld  dword ptr [ecx+8]; // x2   v   x1-xm1
  fsub dword ptr [ecx]; // x2-x0  v   x1-xm1
  fxch st(2);    // x1-m1  v   x2-x0
  fmul c_half;    // c   v   x2-x0
  fxch st(2);    // x2-x0  v   c
  fmul c_half;    // 0.5*(x2-x0) v   c
  fxch st(2);    // c   v   0.5*(x2-x0)
  fst  st(3);    // c   v   0.5*(x2-x0) c
  fadd st(0), st(1);  // w   v   0.5*(x2-x0) c
  fxch st(2);    // 0.5*(x2-x0) v   w   c
  faddp st(1), st(0);  //  v+.5(x2-x0) w   c
  fadd st(0), st(1);  // a   w   c
  fadd st(1), st(0);  // a   b_neg  c
  fmul dword ptr [esp+8]; // a*frac  b_neg  c
  fsubp st(1), st(0);  // a*f-b  c
  fmul dword ptr [esp+8]; // (a*f-b)*f c
  faddp st(1), st(0);  // res-x0/f
  fmul dword ptr [esp+8]; // res-x0
  fadd dword ptr [ecx]; // res
  pop  ecx;
  ret;
 }
}
```

Comments
 from : dmi[at]smartelectronix
 comment : This code produces a nasty buzzing sound. I think there might be a bug somwhere but I haven't found it yet.

 from : hplus-musicdsp[at]mindcontrol.org
 comment : I agree; the output, when plotted, looks like it has overlaid rectangular shapes on it.

 from : robert.bielik@rbcaudio.com
 comment : AHH! True! A bug has sneaked in! Change the row that says:
fsubp st(1), st(0); // a*f-b c
to:
fsubrp st(1), st(0); // a*f-b c

and it should be much better. Although I noticed that a good optimization by the compiler generates nearly as fast a code, but only nearly. This is still about 10% faster.

Hermite interpollation (click this to go back to the index)

References : Posted by various

Notes :
These are all different ways to do the same thing : hermite interpollation. Try'm all and benchmark.

Code :
```
// original
inline float hermite1(float x, float y0, float y1, float y2, float y3)
{
    // 4-point, 3rd-order Hermite (x-form)
    float c0 = y1;
    float c1 = 0.5f * (y2 - y0);
    float c2 = y0 - 2.5f * y1 + 2.f * y2 - 0.5f * y3;
    float c3 = 1.5f * (y1 - y2) + 0.5f * (y3 - y0);

    return ((c3 * x + c2) * x + c1) * x + c0;
}

// james mccartney
inline float hermite2(float x, float y0, float y1, float y2, float y3)
{
    // 4-point, 3rd-order Hermite (x-form)
    float c0 = y1;
    float c1 = 0.5f * (y2 - y0);
    float c3 = 1.5f * (y1 - y2) + 0.5f * (y3 - y0);
    float c2 = y0 - y1 + c1 - c3;

    return ((c3 * x + c2) * x + c1) * x + c0;
}

// james mccartney
inline float hermite3(float x, float y0, float y1, float y2, float y3)
{
        // 4-point, 3rd-order Hermite (x-form)
        float c0 = y1;
        float c1 = 0.5f * (y2 - y0);
        float y0my1 = y0 - y1;
        float c3 = (y1 - y2) + 0.5f * (y3 - y0my1 - y2);
        float c2 = y0my1 + c1 - c3;

        return ((c3 * x + c2) * x + c1) * x + c0;
}

// laurent de soras
inline float hermite4(float frac_pos, float xm1, float x0, float x1, float x2)
{
   const float    c     = (x1 - xm1) * 0.5f;
   const float    v     = x0 - x1;
   const float    w     = c + v;
   const float    a     = w + v + (x2 - x0) * 0.5f;
   const float    b_neg = w + a;

   return ((((a * frac_pos) - b_neg) * frac_pos + c) * frac_pos + x0);
}
```

Comments
from : theguylle
comment : great sources but what is Hermite ?
if you don't describe what is your code made for, you will made a great sources but I don't know why?

cheers Paul

from : bram@musicdsp.org
comment : hermite is interpollation.
have a look around the archive, you'll see that the word 'hermite' is in more than one item ;-)

-bram

from : ronaldowf@sanepar.com.br
comment : Please, would like to know of hermite code it exists in delphi.

thankful

Ronaldo
Cascavel/Paraná/Brasil

from : m.magrini@NOSPAMbad-sector.com
comment : Please,
add, at least, the meaning of each parameter (I mean x, y0, y1,y2, y3)....
m.

from : musicdsp@[remove this]dsparsons.co.uk
comment : Ronaldo, it doesn't take much to translate these to Delphi - for float, either use single or double to your preference!

Looking at the codes, it seems quite clear that the parameters follow a pattern of: Sample Position between middle two samples, then the sample before current, current sample, curernt sample +1, current sample +2.

Lock free fifo (click this to go back to the index)

References : Posted by Timo

Linked file : LockFreeFifo.h (this linked file is included below)

Notes :
Simple implementation of a lock free FIFO.

Comments
from : joshscholar_REMOVETHIS@yahoo.com
comment : There is a good algorithm for a lock free (but multiprocessor safe) FIFO. But the given implimentation is flawed in a number of ways.
This code is not reliable. Two problems on the surface of it:
1. I can easily see that it's possible for two threads/processors to return the same item from the head if the timing is right.
2. there's no interlocked instructions to make sure that changes to the shared variables are globally visible
3. there's not attempt in the code to make sure that data is read in an atomic way, let alone changed in one...

The code is VERY naive

I do have code that works, but it's not so short that will post it in a comment. If anyone needs it they can email me

from : Timo
comment : This is only supposed to work on uniprocessor machines with _one_ reading and _one_ writing thread
assuming that the assignments to read and write idx are simple mov instructions (i.e. atomic). To be sure you'd need to write the update parts in hand-coded asm; never know what the compiler comes up with. The context of this code was omitted (i.e. Bram posted my written in 1 min sketch in a discussion on IRC on a lock-free fifo, not production code).

Linked files

```
#include <vector>
#include <exception>

using std::vector;
using std::exception;

template<class T> class LockFreeFifo
{
public:
 LockFreeFifo (unsigned bufsz) : readidx(0), writeidx(0), buffer(bufsz)
 {}

 T get (void)
 {
  if (readidx == writeidx)
   throw runtime_error ("underrun");

  T result = buffer[readidx];

  if ((readidx + 1) >= buffer.size())
   readidx = 0;
  else
   readidx = readidx + 1;

  return result;
 }

 void put (T datum)
 {
  unsigned newidx;

  if ((writeidx + 1) >= buffer.size())
   newidx = 0;
  else
   newidx = writeidx + 1;

  if (newidx == readidx)
   throw runtime_error ("overrun");

  buffer[writeidx] = datum;

  writeidx = newidx;
 }

private:

 volatile unsigned  readidx, writeidx;
 vector<T> buffer;
};
```

MATLAB-Tools for SNDAN (click this to go back to the index)

References : Posted by Markus Sapp

Linked file : other001.zip

Notes :
(see linkfile)

MIDI note/frequency conversion (click this to go back to the index)

Type : -

References : Posted by tobybear[AT]web[DOT]de

Notes :
I get often asked about simple things like MIDI note/frequency conversion, so I thought I could as well post some source code about this.
The following is Pascal/Delphi syntax, but it shouldn't be a problem to convert it to almost any language in no time.

Uses for this code are mainly for initializing oscillators to the right frequency based upon a given MIDI note, but you might also check what MIDI note is closest to a given frequency for pitch detection etc.
In realtime applications it might be a good idea to get rid of the power and log2 calculations and generate a lookup table on initialization.

A full Pascal/Delphi unit with these functions (including lookup table generation) and a simple demo application can be downloaded here:
http://tobybear.phreque.com/dsp_conv.zip

If you have any comments/suggestions, please send them to: tobybear@web.de

Code :
```
// MIDI NOTE/FREQUENCY CONVERSIONS

const notes:array[0..11] of string= ('C ','C#','D ','D#','E ','F ','F#','G ','G#','A ','A#','B ');
const base_a4=440; // set A4=440Hz

// converts from MIDI note number to frequency
// example: NoteToFrequency(12)=32.703
function NoteToFrequency(n:integer):double;
begin
 if (n>=0)and(n<=119) then
  result:=base_a4*power(2,(n-57)/12)
 else result:=-1;
end;

// converts from MIDI note number to string
// example: NoteToName(12)='C 1'
function NoteToName(n:integer):string;
begin
 if (n>=0)and(n<=119) then
  result:=notes[n mod 12]+inttostr(n div 12)
 else result:='---';
end;

// converts from frequency to closest MIDI note
// example: FrequencyToNote(443)=57 (A 4)
function FrequencyToNote(f:double):integer;
begin
 result:=round(12*log2(f/base_a4))+57;
end;

// converts from string to MIDI note
// example: NameToNote('A4')=57
function NameToNote(s:string):integer;
var c,i:integer;
begin
 if length(s)=2 then s:=s[1]+' '+s[2];
 if length(s)<>3 then begin result:=-1;exit end;
 s:=uppercase(s);
 c:=-1;
 for i:=0 to 11 do
  if notes[i]=copy(s,1,2) then
  begin
   c:=i;
   break
  end;
 try
  i:=strtoint(s[3]);
  result:=i*12+c;
 except
  result:=-1;
 end;
 if c<0 then result:=-1;
end;
```

Comments
from : tobybear@web.de
comment : For the sake of completeness, here is octave fraction notation and pitch class notation:

```
// converts from MIDI note to octave fraction notation
// the integer part of the result is the octave number, where
// 8 is the octave starting with middle C. The fractional part
// is the note within the octave, where 1/12 represents a semitone.
// example: NoteToOct(57)=7.75
function NoteToOct(i:integer):double;
begin
 result:=3+(i div 12)+(i mod 12)/12;
```

```
end;

// converts from MIDI note to pitch class notation
// the integer part of the number is the octave number, where
// 8 is the octave starting with middle C. The
fractional part
// is the note within the octave, where a 0.01 increment is a
// semitone.
// example: NoteToPch(57)=7.09
function NoteToPch(i:integer):double;
begin
 result:=3+(i div 12)+(i mod 12)*0.01;
end;
```

from : kaleja@estarcion.com
comment : I thought most sources gave A-440Hz = MIDI note 69. MIDI 60 = middle C = ~262Hz, A-440 = "A above middle C". Not so?

from : DFL
comment : Kaleja is correct. Here is some C code:

```
double MIDItoFreq( char keynum ) {
  return 440.0 * pow( 2.0, ((double)keynum - 69.0) / 12.0 );
}
```

you can double-check the table here:
http://tomscarff.tripod.com/midi_analyser/midi_note_frequency.htm

Millimeter to DB (faders...) (click this to go back to the index)

References : Posted by James McCartney

Notes :
These two functions reproduce a traditional professional mixer fader taper.

MMtoDB converts millimeters of fader travel from the bottom of the fader for a 100 millimeter fader into decibels. DBtoMM is the inverse.

The taper is as follows from the top:
The top of the fader is +10 dB
100 mm to 52 mm : -5 dB per 12 mm
52 mm to 16 mm : -10 dB per 12 mm
16 mm to 4 mm : -20 dB per 12 mm
4 mm to 0 mm : fade to zero. (in these functions I go to -200dB which is effectively zero for up to 32 bit audio.)

Code :
```
float MMtoDB(float mm)
{
        float db;

        mm = 100. - mm;

        if (mm <= 0.) {
                db = 10.;
        } else if (mm < 48.) {
                db = 10. - 5./12. * mm;
        } else if (mm < 84.) {
                db = -10. - 10./12. * (mm - 48.);
        } else if (mm < 96.) {
                db = -40. - 20./12. * (mm - 84.);
        } else if (mm < 100.) {
                db = -60. - 35. * (mm - 96.);
        } else db = -200.;
        return db;
}

float DBtoMM(float db)
{
        float mm;
        if (db >= 10.) {
                mm = 0.;
        } else if (db > -10.) {
                mm = -12./5. * (db - 10.);
        } else if (db > -40.) {
                mm = 48. - 12./10. * (db + 10.);
        } else if (db > -60.) {
                mm = 84. - 12./20. * (db + 40.);
        } else if (db > -200.) {
                mm = 96. - 1./35. * (db + 60.);
        } else mm = 100.;

        mm = 100. - mm;

        return mm;
}
```

Comments
  from : Christian@savioursofsoul.de
  comment : Pascal Translation...

```
function MMtoDB(Milimeter:Single):Single;
var mm: Single;
begin
  mm:=100-Milimeter;
  if mm = 0 then Result:=10
  else if mm < 48 then Result:=10-5/12*mm;
  else if mm < 84 then Result:=-10-10/12*(mm-48);
  else if mm < 96 then Result:=-40-20./12*(mm-84);
  else if mm < 100 then Result:=-60-35*(mm-96);
  else Result:=-200.;
end;

function DBtoMM(db:Single):Single;
begin
  if db>=10 then result:=0;
  else if db>-10 then result:=-12/5*(db-10);
  else if db>-40 then result:=48-12/10(db+10);
  else if db>-60 then result:=84-12/20(db+40);
  else if db>-200 then result:=96-1/35(db+60);
  else result:=100.;
  Result:=100-Result;
end;
```

Noise Shaping Class (click this to go back to the index)

Type : Dithering with 9th order noise shaping

References : Posted by cshei[AT]indiana.edu

Linked file : NS9dither16.h (this linked file is included below)

Notes :
This is an implemetation of a 9th order noise shaping & dithering class, that runs quite fast (it has one function that uses Intel x86 assembly, but you can replace it with a different rounding function if you are running on a non-Intel platform). _aligned_malloc and _aligned_free require the MSVC++ Processor Pack, available from www.microsoft.com. You can replace them with "new" and "delete," but allocating aligned memory seems to make it run faster. Also, you can replace ZeroMemory with a memset that sets the memory to 0 if you aren't using Win32. Input should be floats from -32768 to 32767 (processS will clip at these points for you, but clipping is bad when you are trying to convert floats to shorts). Note to reviewer - it would probably be better if you put the code in a file such as NSDither.h and have a link to it - it's rather long.

(see linked file)

Comments
from : mail[ns]@mutagene.net
comment : I haven't tried this class out, but it looks like there's a typo in the unrolled loop -- shouldn't it read "c[2]*EH[HistPos+2]"? This might this also account for the 3 clock cycle improvement on a P-III.

```
// This arrangement seems to execute 3 clock cycles faster on a P-III
samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] + c[2]*EH[HistPos+1] +
c[1]*EH[HistPos+1] + c[0]*EH[HistPos];
```

Linked files

```
#pragma once

#include <malloc.h>
#include <rounding.h>

// F-weighted
//static const float or3Fc[3] = {1.623f, -0.982f, 0.109f};
static const float or9Fc[9] = {2.412f, -3.370f, 3.937f, -4.174f, 3.353f, -2.205f, 1.281f, -0.569f,
0.0847f};

// modified-E weighted
//static const float or2MEc[2] = {1.537f, -0.8367f};
//static const float or3MEc[3] = {1.652f, -1.049f, 0.1382f};
//static const float or9MEc[9] = {1.662f, -1.263f, 0.4827f, -0.2913f, 0.1268f, -0.1124f, 0.03252f,
-0.01265f, -0.03524f};

// improved-E weighted
//static const float or5IEc[5] = {2.033f, -2.165f, 1.959f, -1.590f, 0.6149f};
//static const float or9IEc[9] = {2.847f, -4.685f, 6.214f, -7.184f, 6.639f, -5.032f, 3.263f, -
1.632f, 0.4191f};

// Simple 2nd order
//static const float or2Sc[2] = {1.0f, -0.5f};

// Much faster than C's % operator (anyway, x will never be > 2*n in this case so this is very
simple and fast)
// Tell the compiler to try to inline as much as possible, since it makes it run so much faster
since these functions get called at least once per sample
__inline my_mod(int x, int n)
{
 if(x > n) x-=n;
 return x;
}

__inline int round(float f)
{
 int r;
 __asm {
  fld f
  fistp r
 }
 return r;
}

__inline short ltos(long l)
{
 return (short)((l==(short)l) ? l : (l>>31)^0x7FFF);
}
```

```
__inline float frand()
{
 // Linear Congruential Method - got it from the book "Algorithms," by Robert Sedgewick
 static unsigned long a = 0xDEADBEEF;

 a = a * 140359821 + 1;
 return a * (1.0f / 0xFFFFFFFF);
}

class NS9dither16
{
public:
 NS9dither16();
 ~NS9dither16();
 short processS(float samp);
 int processI(float samp);
 void reset();

private:
 int order;
 int HistPos;

 float* c; // Coeffs
 float* EH; // Error History
};

__inline NS9dither16::NS9dither16()
{
 order = 9;

 //c=new float[order];  // if you don't have _aligned_malloc
 c = (float*)_aligned_malloc(order*sizeof(float), 16);
 CopyMemory(c, or9Fc, order*sizeof(float));

 //EH = new float[2*order]; // if you don't have _aligned_malloc
 EH = (float*)_aligned_malloc(2*order*sizeof(float), 16);
 ZeroMemory(EH, 2*order*sizeof(float));

 // Start at top of error history - you can make it start anywhere from 0-8 if you really want to
 HistPos=8;
}

__inline NS9dither16::~NS9dither16()
{
 //if(c) delete [] c; // if you don't have _aligned_free
 //if(EH) delete [] EH; // if you don't have _aligned_free
 if(c) _aligned_free(c); // don't really need "if," since it is OK to free null pointer, but still...
 if(EH) _aligned_free(EH);
}

__inline void NS9dither16::reset()
{
 ZeroMemory(EH, 2*order*sizeof(float));
 // Start at top of error history - you can make it start anywhere from 0-8 if you really want to
 HistPos=8;
}

// Force inline because VC++.NET doesn't inline for some reason (VC++ 6 does)
__forceinline short NS9dither16::processS(float samp)
{
 int output;

 /*for(int x=0; x<order; x++)
 {
  //samp -= c[x] * EH[(HistPos+x) % order];
  samp -= c[x] * EH[HistPos+x];
 }*/
 // Unrolled loop for faster execution
 /*samp -= c[0]*EH[HistPos] + c[1]*EH[HistPos+1] + c[2]*EH[HistPos+2] +
   c[3]*EH[HistPos+3] + c[4]*EH[HistPos+4] + c[5]*EH[HistPos+5] +
   c[6]*EH[HistPos+6] + c[7]*EH[HistPos+7] + c[8]*EH[HistPos+8];*/
 // This arrangement seems to execute 3 clock cycles faster on a P-III
 samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
   c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] +
   c[2]*EH[HistPos+1] + c[1]*EH[HistPos+1] + c[0]*EH[HistPos];

 output = round(samp + (frand() + frand() - 1));

 //HistPos =(HistPos+8) % order; // The % operator is really slow
 HistPos = my_mod((HistPos+8), order);
 // Update buffer (both copies)
 EH[HistPos+9] = EH[HistPos] = output - samp;
```

```
 return ltos(output);
}

__forceinline int NS9dither16::processI(float samp)
{
 int output;

 /*for(int x=0; x<order; x++)
 {
  //samp -= c[x] * EH[(HistPos+x) % order];
  samp -= c[x] * EH[HistPos+x];
 }*/
 // Unrolled loop for faster execution
 /*samp -= c[0]*EH[HistPos] + c[1]*EH[HistPos+1] + c[2]*EH[HistPos+2] +
   c[3]*EH[HistPos+3] + c[4]*EH[HistPos+4] + c[5]*EH[HistPos+5] +
   c[6]*EH[HistPos+6] + c[7]*EH[HistPos+7] + c[8]*EH[HistPos+8];*/
 // This arrangement seems to execute 3 clock cycles faster on a P-III
 samp -= c[8]*EH[HistPos+8] + c[7]*EH[HistPos+7] + c[6]*EH[HistPos+6] +
   c[5]*EH[HistPos+5] + c[4]*EH[HistPos+4] + c[3]*EH[HistPos+3] +
   c[2]*EH[HistPos+1] + c[1]*EH[HistPos+1] + c[0]*EH[HistPos];

 output = round(samp + (frand() + frand() - 1));

 //HistPos =(HistPos+8) % order; // The % operator is really slow
 HistPos = my_mod((HistPos+8), order);
 // Update buffer (both copies)
 EH[HistPos+9] = EH[HistPos] = output - samp;

 return output;
}
```

Nonblocking multiprocessor/multithread algorithms in C++ (click this to go back to the index)

Type : queue, stack, garbage collection, memory allocation, templates for atomic algorithms and types

References : Posted by joshscholarREMOVETHIS@yahoo.com

Linked file : ATOMIC.H

Notes :
see linked file...

pow(x,4) approximation (click this to go back to the index)

References : Posted by Stefan Stenzel

Notes :
Very hacked, but it gives a rough estimate of x**4 by modifying exponent and mantissa.

Code :
```
float p4fast(float in)
{
  long *lp,l;

  lp=(long *)(&in);
  l=*lp;

  l-=0x3F800000l; /* un-bias */
  l<<=2;          /* **4 */
  l+=0x3F800000l; /* bias */
  *lp=l;

  /* compiler will read this from memory since & operator had been used */
  return in;
}
```

Reading the compressed WA! parts in gigasampler files (click this to go back to the index)

References : Paul Kellett

Linked file : gigxpand.zip

Notes :
(see linkfile)
Code to read the .WA! (compressed .WAV) parts of GigaSampler .GIG files.
For related info on reading .GIG files see http://www.linuxdj.com/evo

Real basic DSP with Matlab (+ GUI) ... (click this to go back to the index)

Type : Like effects racks, made with Matlab !

References : Posted by guillaume[DOT]carniato[AT]meletu[DOT]univ-valenciennes[DOT]fr

Linked file : http://www.xenggeng.fr.st/ici/guitou/Matlab Music.zip

Notes :
You need Matlab v6.0 or more to run this stuff...

Code :
```
take a look at http://www.xenggeng.fr.st/ici/guitou/Matlab Music.zip

I'm now working on a Matlab - sequencer, which will certainly use 'Matlab Music'. I'm interested in
integrating WaveWarp in this project; it's a toolbox that allow you to make real time DSP with Matlab.

If you're ok to improve this version (add effects, improve effects quality,anything else...) let's go ! Email
me if you're interested in developing this beginner work...
```

real value vs display value (click this to go back to the index)

Type : Macro

References : Posted by emil[AT]arpanet[DOT]no

Notes :

REALVAL converts the vst param at given ranges to a display value.
VSTVAL does the opposite.
a = start
b = end

Code :
```
#define REALVAL(a, b, vstval) (a + (vstval)*(b-a))
#define VSTVAL(a, b, realval) ((realval-a)/(b-a))
```

Comments
from : bekkah@web.de
comment : Why da hell do you use Makros???

BTW: I'll post my mapper class in a few days here, which does all this in a much more convenient way.

from : kaleja@estarcion.com
comment : See http://www.u-he.com/vstsource/archive.php?classid=2#16 for my solutions to this problem.

Really fast x86 floating point sin/cos (click this to go back to the index)

References : Posted by rerdavies[AT]msn[DOT]com

Linked file : sincos.zip

Notes :
Frightful code from the Intel Performance optimization front. Not for the squeamish.

The following code calculates sin and cos of a floating point value on x86 platforms to 20 bits precision with 2 multiplies and two adds. The basic principle is to use sin(x+y) and cos(x+y) identities to generate the result from lookup tables. Each lookup table takes care of 10 bits of precision in the input. The same principle can be used to generate sin/cos to full (! Really. Full!) 24-bit float precision using two 8-bit tables, and one 10 bit table (to provide guard bits), for a net speed gain of about 4x over fsin/fcos, and 8x if you want both sin and cos. Note that microsoft compilers have trouble keeping doubles aligned properly on the stack (they must be 8-byte aligned in order not to incur a massive alignment penalty). As a result, this class should NOT be allocated on the stack. Add it as a member variable to any class that uses it.

e.g.
```
 class CSomeClass {
    CQuickTrig m_QuickTrig;
  ...
    mQuickTrig.QuickSinCos(dAngle,fSin,fCos);
  ...
  }
```

Code :
```
(see attached file)
```

resampling (click this to go back to the index)

Type : linear interpolated aliased resampling of a wave file

References : Posted by mail@mroc.de

Notes :
som resampling stuff. the code is heavily used in MSynth, but do not lough about ;-)

perhaps, prefiltering would reduce aliasing.

Code :
```
signed short* pSample = ...;
unsigned int sampleLength = ...;

// stretch sample to length of one bar...
float playPosDelta = sampleLength / ( ( 240.0f / bpm ) * samplingRate );

// requires for position calculation...
float playpos1 = 0.0f;
unsigned int iter = 0;

// required for interpolation...
unsigned int i1, i2;

float* pDest = ....;
float* pDestStop = pDest + len;
for( float *pt=pDest;pt<pDestStop;++pt )
{
 // linear interpolation...
 i1 = (unsigned int)playpos;
 i2 = i1 + 1;
 (*pt) = ((pSample[i2]-pSample[i1]) * (playpos - i1) + pSample[i1]);

 // position calculation preventing float sumation error...
 playpos1 = (++iter) * playposIncrement;
}
...
```

Saturation (click this to go back to the index)

Type : Waveshaper

References : Posted by Bram

Notes :
when the input is below a certain threshold (t) these functions return the input, if it goes over that threshold, they return a soft shaped saturation.
Neigther claims to be fast ;-)

Code :
```
float saturate(float x, float t)
{
    if(fabs(x)<t)
        return x
    else
    {
        if(x > 0.f);
            return t + (1.f-t)*tanh((x-t)/(1-t));
        else
            return -(t + (1.f-t)*tanh((-x-t)/(1-t)));
 }
}

or

float sigmoid(x)
{
    if(fabs(x)<1)
        return x*(1.5f - 0.5f*x*x);
    else
        return x > 0.f ? 1.f : -1.f;
}

float saturate(float x, float t)
{
    if(abs(x)<t)
        return x
    else
    {
        if(x > 0.f);
            return t + (1.f-t)*sigmoid((x-t)/((1-t)*1.5f));
        else
            return -(t + (1.f-t)*sigmoid((-x-t)/((1-t)*1.5f)));
    }
}
```

Comments
from : terry@yahoo.com
comment : But My question is
BUT HAVE YOU TRIED YOUR CODE!!!!!!!!!!!!!!!!????
I think no, 'cos give a compiling error.
the right (for sintax) version is this:

```
float sigmoid(float x)
{
    if(fabs(x)<1)
        return x*(1.5f - 0.5f*x*x);
    else
        return x > 0.f ? 1.f : -1.f;
}

float saturate(float x, float t)
{
    if(abs(x)<t)
        return x;
    else
    {
        if(x > 0.f)
            return t + (1.f-t)*sigmoid((x-t)/((1-t)*1.5f));
        else
            return -(t + (1.f-t)*sigmoid((-x-t)/((1-t)*1.5f)));
    }
}
```

from : imbeachhunt@hotmail.com
comment : except for the missing parenthesis of course =)
the first line of saturate should be either

if(fabs(x)) return x;

or

if(abs(x)) return x;

depending on whether you're looking at the first or second saturate function (in the orig post)

Sin, Cos, Tan approximation (click this to go back to the index)

References : http://www.wild-magic.com

Linked file : approx.h (this linked file is included below)

Notes :
Code for approximation of cos, sin, tan and inv sin, etc.
Surprisingly accurate and very usable.

[edit by bram]
this code is taken literaly from
http://www.wild-magic.com/SourceCode.html
Go have a look at the MgcMath.h and MgcMath.cpp files in their library...
[/edit]

Comments
from : asynth@io.com
comment : It'd be nice to have a note on the domain of these functions. I assume Sin0 is meant to be used about zero and Sin1 about 1. But a note to that effect would be good.
Thanks,

james mccartney

from : mcodespam@gmx.net
comment : Sin0 is faster but less accurate than Sin1, same for the other pairs. The domains are:

Sin/Cos [0, pi/2]
Tan [0,pi/4]
InvSin/Cos [0, 1]
InvTan [-1, 1]

This comes from the original header file.

Linked files

```
//----------------------------------------------------------------------
Real Math::FastSin0 (Real fAngle)
{
    Real fASqr = fAngle*fAngle;
    Real fResult = 7.61e-03f;
    fResult *= fASqr;
    fResult -= 1.6605e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastSin1 (Real fAngle)
{
    Real fASqr = fAngle*fAngle;
    Real fResult = -2.39e-08f;
    fResult *= fASqr;
    fResult += 2.7526e-06f;
    fResult *= fASqr;
    fResult -= 1.98409e-04f;
    fResult *= fASqr;
    fResult += 8.3333315e-03f;
    fResult *= fASqr;
    fResult -= 1.666666664e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastCos0 (Real fAngle)
{
    Real fASqr = fAngle*fAngle;
    Real fResult = 3.705e-02f;
    fResult *= fASqr;
    fResult -= 4.967e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastCos1 (Real fAngle)
```

```
{
    Real fASqr = fAngle*fAngle;
    Real fResult = -2.605e-07f;
    fResult *= fASqr;
    fResult += 2.47609e-05f;
    fResult *= fASqr;
    fResult -= 1.3888397e-03f;
    fResult *= fASqr;
    fResult += 4.16666418e-02f;
    fResult *= fASqr;
    fResult -= 4.999999963e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastTan0 (Real fAngle)
{
    Real fASqr = fAngle*fAngle;
    Real fResult = 2.033e-01f;
    fResult *= fASqr;
    fResult += 3.1755e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastTan1 (Real fAngle)
{
    Real fASqr = fAngle*fAngle;
    Real fResult = 9.5168091e-03f;
    fResult *= fASqr;
    fResult += 2.900525e-03f;
    fResult *= fASqr;
    fResult += 2.45650893e-02f;
    fResult *= fASqr;
    fResult += 5.33740603e-02f;
    fResult *= fASqr;
    fResult += 1.333923995e-01f;
    fResult *= fASqr;
    fResult += 3.333314036e-01f;
    fResult *= fASqr;
    fResult += 1.0f;
    fResult *= fAngle;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastInvSin (Real fValue)
{
    Real fRoot = Math::Sqrt(1.0f-fValue);
    Real fResult = -0.0187293f;
    fResult *= fValue;
    fResult += 0.0742610f;
    fResult *= fValue;
    fResult -= 0.2121144f;
    fResult *= fValue;
    fResult += 1.5707288f;
    fResult = HALF_PI - fRoot*fResult;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastInvCos (Real fValue)
{
    Real fRoot = Math::Sqrt(1.0f-fValue);
    Real fResult = -0.0187293f;
    fResult *= fValue;
    fResult += 0.0742610f;
    fResult *= fValue;
    fResult -= 0.2121144f;
    fResult *= fValue;
    fResult += 1.5707288f;
    fResult *= fRoot;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastInvTan0 (Real fValue)
{
    Real fVSqr = fValue*fValue;
    Real fResult = 0.0208351f;
    fResult *= fVSqr;
    fResult -= 0.085133f;
    fResult *= fVSqr;
```

```
    fResult += 0.180141f;
    fResult *= fVSqr;
    fResult -= 0.3302995f;
    fResult *= fVSqr;
    fResult += 0.999866f;
    fResult *= fValue;
    return fResult;
}
//----------------------------------------------------------------------
Real Math::FastInvTan1 (Real fValue)
{
    Real fVSqr = fValue*fValue;
    Real fResult = 0.0028662257f;
    fResult *= fVSqr;
    fResult -= 0.0161657367f;
    fResult *= fVSqr;
    fResult += 0.0429096138f;
    fResult *= fVSqr;
    fResult -= 0.0752896400f;
    fResult *= fVSqr;
    fResult += 0.1065626393f;
    fResult *= fVSqr;
    fResult -= 0.1420889944f;
    fResult *= fVSqr;
    fResult += 0.1999355085f;
    fResult *= fVSqr;
    fResult -= 0.3333314528f;
    fResult *= fVSqr;
    fResult += 1.0f;
    fResult *= fValue;
    return fResult;
}
//----------------------------------------------------------------------
```